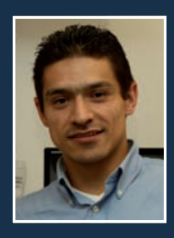
Programación Orientada a Objetos usando Java

ECOE EDICIONES

Héctor Arturo Flórez Fernández



HÉCTOR ARTURO FLÓREZ FERNÁNDEZ

Ingeniero electrónico e ingeniero de sistemas de la Universidad El Bosque; magíster en Ciencias de la Información y las Comunicaciones de la Universidad Distrital Francisco José de Caldas; especialista en Alta Gerencia y magíster en Gestión de Organizaciones de la Universidad Militar Nueva Granada; estudiante de Doctorado en Ingeniería en la Universidad de los Andes.

Programación Orientada a Objetos usando Java

Héctor Arturo Flórez Fernández

Catalogación en la publicación – Biblioteca Nacional de Colombia

Flórez Fernández, Héctor Arturo
Programación orientada a objetos usando java / Héctor Arturo Flórez Fernández. -- 1a. ed. -- Bogotá : Ecoe Ediciones, 2012.
412 p. – (Ingeniería y arquitectura. Informática)

Incluye bibliografía
ISBN 978-958-648-796-2

1. Programación orientada a objetos (Computación) 2. Java (Lenguaje de programación de computadores) I. Título II. Serie

CDD: 005.133 ed. 21 CO-BoBN– a819455

Colección: Ingeniería y arquitectura
Área: Ingeniería

Primera edición: Bogotá, D.C., 2012

ISBN: 978-958-648-796-2

Reservados todos los derechos
© Héctor Arturo Flórez Fernández
e-mail: hectorarturo@yahoo.com

© Ecoe Ediciones
E-mail: correo@ecoeediciones.com
www.ecoeediciones.com
Carrera 19 No. 63C-32, Pbx. 2481449, fax. 3461741

Coordinación editorial: Alexander Acosta Quintero
Diseño y diagramación: Yolanda Madero Tiria
Diseño de carátula: Edwin Penagos Palacio
Impresión: Imagen editorial impresores
imagenimvega@yahoo.com

Impreso y hecho en Colombia.

Tabla de contenido

Índice de Figuras

Índice de Tablas

Introducción

El libro Programación Orientada a Objetos usando Java ofrece al lector una exposición clara y suficiente de los conceptos básicos de programación orientada a objetos y desarrollo de aplicaciones mediante el lenguaje de programación Java.

El documento expone con una gran cantidad de ejemplos y demostraciones, las diferentes características de la Programación Orientada a Objetos (POO), y su uso, mediante el lenguaje Java, además de orientar el desarrollo mediante arquitecturas, patrones y buenas prácticas en el desarrollo de aplicaciones.

Aquí se ofrecen explicaciones de conceptos básicos de programación y conceptos de programación orientada a objetos; del desarrollo orientado a objetos con base en arquitectura de tres capas, con acceso a repositorios de datos mediante archivos planos, serializables y bases de datos; conceptos básicos de computación gráfica, patrones de diseño, procesos multitarea, comunicaciones en red, captura de vídeo mediante *API* de multimedia, carga dinámica de clases y librerías, entre otros.

CAPÍTULO 1

Introducción al lenguaje de programación Java

Java fue creado en 1991 por Sun Microsystems con el fin de elaborar un lenguaje de programación destinado a electrodomésticos. Debido a la existencia de distintos tipos de procesadores y a los continuos cambios, se generó la necesidad de hacer una herramienta independiente de la clase de procesador. Entonces, Sun Microsystems creó una aplicación neutra que no dependía del tipo de electrodoméstico. Esta aplicación se ejecutaba a través de una máquina hipotética o virtual denominada "*Java Virtual Machine, JVM*" o máquina virtual de Java. La *JVM* interpretaba el código neutro convirtiéndolo a código ensamblador que podía ser interpretado por el procesador utilizado.

A finales de 1995, Java fue introducido como lenguaje de programación para computadores. La versión Java 1.1, apareció a principios de 1997 mejorando sustancialmente la versión original del lenguaje. La versión Java 1.2, aparece más tarde y fue renombrada a Java 2, a finales de 1998.La Java 1.5 se presenta en 2005. Actualmente se desarrolla sobre la versión Java 1.6 a partir de 2006.

Al desarrollar en Java, cualquier aplicación, se cuenta con un gran número de clases que hacen parte del lenguaje de Java conocido como "*API* o *Application Programming Interface*". El *API* de Java se organiza por paquetes que hacen referencia a contenedores de clases. La funcionalidad de cada concepto lo proveen las respectivas clases.

Java incorpora en el propio lenguaje muchos aspectos. Por tal motivo, gran parte de la comunidad académica opina que Java es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo de desarrollo.

El principal objetivo del lenguaje Java es convertirse en el lenguaje universal de desarrollo. Java es un lenguaje muy completo. Java 1.0 tenía 12 paquetes, Java 1.1 tenía 23paquetes y Java 1.2 tenía 59paquetes. Para 2010, la versión Java 1.6 cuenta con 203 paquetes y 3.793 clases, lo que indica la gran potencia del lenguaje.

Las aplicaciones desarrolladas en Java presentan diversas ventajas. La ejecución de programas en Java tiene muchas posibilidades: ejecución como aplicación independiente denominada "*Stand-alone Application*" o aplicación de escritorio, ejecución como "*applet*", la cual es una aplicación especial que se ejecuta dentro de un navegador de Internet, ejecución como "*servlet*", el cual permite proveer servicios a aplicaciones *web* y se ejecuta siempre en un servidor de Internet. Además, las aplicaciones desarrolladas en Java son portables, lo que significa que se pueden compilar y ejecutar sobre cualquier sistema operativo porque depende directamente de su *JMV*.

El crecimiento de Java ha sido tan rápido que Sun Microsystems ha desarrollado soluciones personalizadas para cada ámbito tecnológico, agrupando cada uno de esos ámbitos en una edición distinta que son:

- *J2SE* (*Java 2 Stantard Edition*)
- *J2EE* (*Java 2 Enterprise Edition*)
- *J2ME* (*Java 2 Micro Edition*)

Existen distintas aplicaciones que permiten desarrollar código Java. La compañía Sun Microsystems, creadora de Java, distribuye gratuitamente el *Java* (TM) *Development Kit* (*JDK*)[1] que se compone de un conjunto de programas y librerías que permiten desarrollar,

[1] http://www.oracle.com/es/technologies/java/index.html

compilar y ejecutar programas en Java. Incorpora además, la posibilidad de ejecutar parcialmente el programa mediante un proceso denominado "*Debugger*" o "Depuración", deteniendo la ejecución en el punto deseado y estudiando en cada momento, el valor de cada una de las variables.

Existe también una versión reducida del *JDK*, denominada "*Java Runtime Environment* (*JRE*)"[2] destinada únicamente a ejecutar código Java, es decir, que con el *JRE* no es posible compilar el código.

Por otro lado, los *Integrated Development Environment, (IDE)*, son entornos de desarrollo integrados como Eclipse IDE[3], Netbeans[4], JCreator[5], JBuilder[6], etc. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar depuración gráficamente, frente a la versión que incorpora el *JDK* basada en la utilización de una consola bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa.

1.1 Compilador de Java

El compilador es una herramienta de desarrollo incluidas en el *JDK*. Realiza un análisis de sintaxis del código fuente escrito en archivos fuente de Java que poseen extensión "***.java***". Si no se encuentran errores en el código se genera un archivo compilado por cada archivo fuente con extensión "***.class***". En caso de existir errores, no se crea el archivo compilado y se presenta información del error. En el *JDK* dicho compilador se llama "***javac.exe***".

[2] http://www.oracle.com/es/technologies/java/index.html

[3] http://www.eclipse.org/

[4] http://netbeans.org/

[5] http://www.jcreator.com/

[6] http://www.embarcadero.com/products/jbuilder

1.2 *Java Virtual Machine*

Java Virtual Machine, JVM o máquina virtual de Java es un proyecto que se puede instalar en cualquier sistema operativo y permite que un archivo compilado se ejecute sin requerir cambios, independientemente del procesador que posea el computador. La clave consistió en desarrollar un código "neutro" el cual es interpretado por la *JVM* convirtiéndolo a código particular de la *CPU* utilizada. Con la *JVM* se evita tener que realizar un programa diferente para cada *CPU* o plataforma.

La máquina virtual de Java es el intérprete de Java. Ejecuta los "*bytecodes*" que son los mismos archivos compilados con extensión "***.class***" creados por el compilador de Java "***javac.exe***". Tiene numerosas opciones, entre las que se destaca la posibilidad de utilizar el denominado *JIT* (*Just-In-Time Compiler*), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

1.3 *Garbage collector*

El "*garbage collector"* o recolector de basura es una aplicación que hace parte de la máquina virtual de Java y se encarga de identificar los objetos que han perdido su referencia en tiempo de ejecución, es decir, recoge los objetos que no están siendo utilizados y los destruye de forma automática con el fin de liberar memoria dinámica.

Es posible usar explícitamente el recolector de basura mediante la sentencia "*System.gc()*". También es posible retirar la referencia del objeto asignándole el valor "*null*", de esta forma se permite que el recolector de basura identifique dicho objeto y lo destruya, liberando esa porción de memoria dinámica.

1.4 Variables *Path* y *Classpath*

El desarrollo y ejecución de aplicaciones en Java exige que las herramientas para compilar y ejecutar se encuentren accesibles. El computador solo es capaz de ejecutar los programas que se

encuentran en los directorios sindicados en la variable *Path* del mismo. Si se desea compilar o ejecutar código en Java, el directorio donde se encuentran estos programas (***java.exe*** y ***javac.exe***) deberá encontrarse en el *Path*.

Java utiliza además una nueva variable de entorno denominada *Classpath*, la cual determina dónde buscar tanto las clases o librerías de Java que corresponden al *API* de Java. A partir de la versión 1.1.4 del *JDK* no es necesario indicar esta variable, salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho *JDK*. La variable *Classpath* puede incluir la ruta de directorios o archivos "***.zip***" o "***.jar***" en los que se encuentren los archivos "***.class***". En el caso de los archivos ***".zip***" hay que tener en cuenta que los archivos incluidos no estén comprimidos. En el caso de archivos "***.jar***" existe una herramienta ***jar.exe***, incorporada en el *JDK*, que permite generar estos archivos a partir de los archivos compilados.

Los comandos para configurar las variables *Path y Classpath* son:

```
set JAVAPATH=C:\jdk1.6
set PATH=.;%JAVAPATH%\bin;%PATH%
set CLASSPATH=.\;%JAVAPATH%\lib\classes.zip;%CLASSPATH%
```

Estas líneas son validas en el caso de que el *JDK* estuviera situado en el directorio *C:\jdk1.6*.

También es posible utilizar la opción *–classpath* en el momento de llamar al compilador ***javac.exe*** o al intérprete ***java.exe***. En este caso los archivos "***.jar***" deben ponerse con el nombre completo en el *Classpath*: no basta poner el *Path* o directorio en el que se encuentra. Por ejemplo, si se desea compilar y ejecutar el archivo*principal.java*, y este necesitara la librería de clases *C:\MiProyecto\MisClases.jar*, además de las incluidas en el *Classpath*, la forma de compilar y ejecutar sería:

```
javac -classpath .\; C:\MiProyecto\MisClases.jar principal.java
java -classpath .\; C:\MiProyecto\MisClases.jar principal
```

Se aconseja consultar la ayuda correspondiente a la versión que se esté utilizando, debido a que existen pequeñas variaciones entre las distintas versiones del *JDK*.

1.5 Primer programa en Java

Para ejecutar un programa en Java es necesario crear una clase principal, que contenga un método denominado "*main*" con visibilidad "*public*" y tipo "*static*" (los conceptos de clase, método, *public* y *static* se explican en el capítulo 3). Este será el primer método que se ejecuta al iniciar la aplicación.

La implementación es la siguiente.

```
public class Principal {

   public static void main(String[] args) {
       System.out.println("HOLA MUNDO");
   }
}
```

Esta aplicación permite imprimir el mensaje "HOLA MUNDO" en salida estándar o también llamada consola.

CAPÍTULO 2

Conceptos básicos de programación

Para desarrollar cualquier aplicación es necesario utilizar diferentes conceptos, que se ofrecen en cualquier lenguaje y paradigma.

2.1 Operadores

En los lenguajes de programación existen diferentes tipos de operadores que permiten realizar diferentes procesos. Estos operadores se clasifican en aritméticos, de asignación, lógicos y de comparación.

2.1.1 Operadores aritméticos

Los operadores aritméticos resuelven las operaciones básicas de suma, resta, multiplicación, división y módulo, el cual entrega el residuo de la división.

Estos operadores se pueden aplicar a variables numéricas. Sin embargo, el operador suma se puede aplicar a variables que contengan cadenas de caracteres generando concatenación de la información.

Existen operadores especiales de incremento y decremento que permiten sumar y restar el valor 1 respectivamente a la variable.

Los símbolos de las operaciones aritméticas se presentan en la Tabla 1.

Tabla 1. Operadores aritméticos

Operador	Símbolo	Implementación
Suma	+	B+C
Resta	-	B-C
Multiplicación	*	B*C
División	/	B/C
Módulo	%	B%C
Incremento	++	A++
Decremento	--	A--

2.1.2 Operadores de asignación

Los operadores de asignación permiten depositar un valor en una variable. En muchos casos es necesario realizar una operación aritmética de dos variables, cuyo resultado debe depositarse en una de esas variables. Para estos casos, también se puede aplicar un operador especial. Los símbolos de los operadores de asignación se presentan en la Tabla 2.

Tabla 2. Operadores de asignación

Operador	Símbolo	Implementación
Asignación	=	A=10
Asignación con Suma	+=	A+=B ⬄ A=A+B
Asignación con Resta	-=	A-=B ⬄ A=A-B
Asignación con Multiplicación	*=	A*=B ⬄ A=A*B
Asignación con División	/=	A/=B ⬄ A=A/B
Asignación con Módulo	%=	A%=B ⬄ A=A%B

2.1.3 Operadores lógicos

Los operadores lógicos resuelven expresiones booleanas. El resultado de estas operaciones será siempre verdadero "*true*" o falso "*false*". Las operaciones booleanas básicas son *AND*, *OR* y *NOT*.

La operación lógica *NOT*, se aplica siempre sobre una premisa, que en un lenguaje de programación estará descrita en una variable booleana. Esta operación consiste en cambiar el valor de la premisa de falso a verdadero y viceversa. El comportamiento de esta operación lógica se define en la Tabla 3.

Tabla 3. Tabla de verdad de la operación lógica *NOT*

Entrada	NOT
Falso	Verdadero
Verdadero	Falso

La operación lógica *AND*, indica que la salida será verdadera si y solo si, todas sus entradas son verdaderas. El comportamiento de esta operación lógica se define en la Tabla 4.

Tabla 4. Tabla de verdad de la operación lógica *AND*

Entrada 1	Entrada 2	AND
Falso	Falso	Falso
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Verdadero	Verdadero	Verdadero

La operación lógica *OR*, indica que la salida será falsa si y solo si, todas sus entradas son falsas. El comportamiento de esta operación lógica se define en la Tabla 5.

Tabla 5. Tabla de verdad de la operación lógica *OR*

Entrada 1	Entrada 2	NOT
Falso	Falso	Falso
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Verdadero	Verdadero	Verdadero

La operación lógica *XOR*, indica que la salida será verdadera si el número de entradas verdaderas es impar. El comportamiento de esta operación lógica se define en la Tabla 6.

Tabla 6. Tabla de verdad de la operación lógica *XOR*

Entrada 1	Entrada 2	NOT
Falso	Falso	Falso
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Verdadero	Verdadero	Falso

Los símbolos de las operaciones lógicas se presentan en la Tabla 7.

Tabla 7. Operadores lógicos

Operador	Símbolo	Implementación	Descripción
NOT	`!`	`!A`	Negación
AND	`&&`	`A && B`	Si A es falso, no se evalúa B
OR	`||`	`A || B`	Si A es verdadero, no se evalúa B
AND	`&`	`A & B`	Si A es falso, siempre evalúa B
OR	`|`	`A | B`	Si A es verdadero, siempre evalúa B

2.1.4 Operadores de comparación

Los operadores de comparación permiten la verificación de dos variables, determinando si una de ellas es mayor, igual, menor o diferente de la otra. El resultado de estas operaciones será siempre verdadero "*true*" o falso "*false*". Los símbolos de las operaciones de comparación se presentan en la Tabla 8.

Tabla 8. Operadores de comparación

Operador	Símbolo	Implementación
Igual	==	A==B
Diferente	!=	A!=B
Mayor	>	A>B
Menor	<	A<B
Mayor o Igual	>=	A>=B
Menor o Igual	<=	A<=B

2.1.5 Operadores a nivel de *bits*

Los operadores a nivel de *bits* permiten aplicar operaciones a los *bits* de los datos. Los símbolos de los operadores a nivel de *bits* se presentan en la Tabla 9.

Tabla 9. Operadores a nivel de *bits*

Símbolo	Implementación	Descripción
>>	A >> B	Desplaza los *bits* de A a la derecha una distancia B
<<	A << B	Desplaza los *bits* de A a la izquierda una distancia B
&	A & B	AND a nivel de *bits*
|	A | B	OR a nivel de *bits*
^	A ^ B	XOR a nivel de *bits*
~	~A	Complemento a nivel de *bits*

2.2 Tipos primitivos de datos

Todo programa requiere la creación de variables las cuales permitirán el almacenamiento de información temporal en memoria dinámica. Cada una de las variables creadas, deben tener características definidas por tipos de datos para poder almacenar información. La Tabla 10 presenta los tipos primitivos de datos y sus características principales.

Tabla 10. Tipos primitivos de datos

Tipo	Descripción	Tamaño	Posibles valores
byte	Número entero con signo	1 *byte*	-128 a 127
short	Número entero con signo	2 *bytes*	-32768 a 32767
int	Número entero con signo	4 *bytes*	-2.147.483.648 a 2.147.483.647
long	Número entero con signo	8 *bytes*	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	Número flotante con signo, con precisión simple	4 *bytes*	-3.402823E38 a -1.401298E-45 y 1.401298E-45 a 3.402823E38
double	Número flotante con signo, con precisión doble	8 *bytes*	-1.79769313486232E308 a -4.94065645841247E-324 y 4.94065645841247E-324 a 1.79769313486232E308

char	Carácter de código ASCII	2 *bytes*	Símbolos Unicode
boolean	Cantidad booleana	1 *byte*	*true, false*

2.2.1 Conversión de tipos primitivos de datos

La conversión entre tipos primitivos se realiza de modo automático en conversiones implícitas de un tipo a otro de más precisión, por ejemplo, de *int* a *long* o *float* a *double*. Estas conversiones se hacen necesarias en el momento de involucrar variables de diferentes tipos en expresiones matemáticas. Así mismo, en el momento en que se requiere ejecutar sentencias de asignación en las que el término izquierdo tiene un tipo diferente al resultado de evaluación en el término derecho. Las siguientes sentencias ejemplifican el uso de conversión implícita.

```
int dato1=1000;
int dato2=10000;
long resultado;
resultado=dato1*dato2;
```

Nótese en el ejemplo anterior, que *dato1* y *dato2* son tipo *int*, entonces su resultado es tipo *int*. Sin embargo, el tipo *long* tiene mayor capacidad que el tipo *int*, entonces, de forma implícita se realiza la conversión de tipo.

Las conversiones de un tipo de mayor precisión a otro de menor precisión requieren una sentencia explícita, debido a que estas conversiones pueden generar pérdida de información. A estas conversiones explícitas de tipo se les llama "*cast*" o "*casting*". El "*cast*" se hace colocando el tipo al que se desea transformar entre paréntesis, previo a la expresión a la que va a realizarse la conversión.

Las siguientes sentencias ejemplifican el uso de conversión implícita.

```
long dato1=1000;
long dato2=10000;
int resultado;
resultado=(int)(dato1*dato2);
```

Nótese en el ejemplo anterior, que *dato1* y *dato2* son tipo *long*, entonces su resultado es tipo *long*. Entonces como el tipo *long* tiene mayor capacidad que el tipo *int*, es necesario realizar la conversión de tipo de forma explícita para que no se presenten errores en la compilación de la aplicación.

2.3 Estructuras de programación

Las estructuras de programación o también llamadas estructuras de control permiten implementar procesos, tomar decisiones y realizar procesos con varias repeticiones.

2.3.1 Sentencias

Una expresión es un conjunto de variables unidas por operadores. Equivalen a instrucciones que el computador interpreta para realizar un proceso determinado. Una sentencia es una expresión que tiene al final punto y coma (;). Es posible incluir varias sentencias en una línea, sin embargo, se considera una buena práctica utilizar una línea para cada sentencia. Las siguientes líneas son ejemplos de sentencias en un programa.

```
int a;
int b;
int c;
b=10;
c=20;
a=b+c;
```

2.3.2 Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de la aplicación. Los comentarios son útiles para documentar el código implementado.

Los comentarios se realizan de dos formas. La primera consiste en colocar el símbolo "*//*" en una línea de código y en seguida el texto del comentario. La segunda consiste en incluir el símbolo "*/**" al inicio del comentario y el símbolo "**/*" al final del comentario. Esta segunda forma permite hacer comentarios en varias líneas de código.

```
//Este es un comentario en una línea de código

/*
Este es un comentario
En diferentes líneas de código
*/
```

2.3.3 Estructura de condición *if*

La estructura de condición "*if*" se compone de una condición la cual siempre debe arrojar un valor booleano, es decir, verdadero o falso. Esta condición debe encontrarse entre paréntesis. Esta permite ejecutar un conjunto de instrucciones si se cumple la condición establecida. Este conjunto de instrucciones debe estar incluido entre los símbolos "*{*" y "*}*". Sin embargo, si solo se desea ejecutar una instrucción no es necesario incluir los símbolos "*{*" y "*}*". La sintaxis de esta sentencia es:

```
if(condición){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
}
```

O también:

```
if(condición)
   instrucción;
```

Ejemplo:

```
/* Imprime en consola el mensaje entre paréntesis en la
instrucción System.out.println() */
if(n==5){
   System.out.println("El valor de n es cinco");
}
```

2.3.4 Estructura de condición *if else*

La estructura de condición "*if*" "*else*" se compone de una condición, la cual siempre debe arrojar un valor booleano, es decir, verdadero o falso. Esta condición debe encontrarse entre paréntesis. Esta permite ejecutar un conjunto de instrucciones si se cumple la condición establecida y permite ejecutar otro conjunto de instrucciones diferentes si no se cumple la condición establecida. Este conjunto de instrucciones debe estar incluido entre los símbolos "*{*" y "*}*". La sintaxis de esta sentencia es:

```
if(condición){
   instrucción 1.1;
   instrucción 1.2;
   ..
   instrucción 1.n;
}else{
   instrucción 2.1;
   instrucción 2.2;
   ..
   instrucción 2.n;
}
```

Ejemplo:

```
if(n==5){
   System.out.println("El valor de n es cinco");
}else{
   System.out.println("El valor de n es diferente de cinco");
}
```

La estructura de condición "*if* " "*else*", se puede implementar con una sintaxis avanzada que permita la codificación en una sola línea. Esta sintaxis se debe aplicar si y solo si, se cuenta con un solo proceso en el "*if* " y el "*else*" y tienen el mismo comportamiento. La sintaxis de esta sentencia es:

```
(condición) ? instrucción 1 : instrucción 2;
```

Ejemplo:

```
System.out.println((n==5) ? "El valor de n es cinco" : "El valor
de n es diferente de cinco");
```

2.3.5 Estructura de condición *if else if*

La estructura de condición "*if*" "*else if*" se compone de múltiples condiciones, las cuales siempre deben arrojar un valor booleano, es decir, verdadero o falso. Estas condiciones deben encontrarse entre paréntesis. Esta permite ejecutar un conjunto de instrucciones si se cumple la condición establecida y permite ejecutar otro conjunto de instrucciones diferentes, si se cumple la otra condición establecida y así sucesivamente. Este conjunto de instrucciones debe estar incluido entre los símbolos "***{***" y "***}***". Esta estructura permite tener opcionalmente, al final una estructura *else*, la cual se ejecuta si ninguna de las condiciones fueron verdaderas. Si una condición es verdadera se ejecutan las instrucciones correspondientes y no consulta las siguientes condiciones. La sintaxis de esta sentencia es:

```
if(condición 1){
   instrucción 1.1;
   instrucción 1.2;
   ..
   instrucción 1.n;
}else if(condición 2){
   instrucción 2.1;
   instrucción 2.2;
   ..
   instrucción 2.n;
}else if(condición 3){
   instrucción 3.1;
   instrucción 3.2;
   ..
   instrucción 3.n;
}
```

Ejemplo:

```
if(n==1){
   System.out.println("El valor de n es uno");
}else if(n==2){
   System.out.println("El valor de n es dos");
}else if(n==3){
   System.out.println("El valor de n es tres");
}else{
   System.out.println("El valor de n es mayor que tres");
}
```

2.3.6 Estructura de condición *switch case*

La estructura de condición "*switch*" "*case*" se compone de múltiples condiciones sobre una misma variable, las cuales siempre deben arrojar un valor booleano, es decir, verdadero o falso. La variable sujeto de la condición debe encontrarse entre paréntesis, posterior a la sentencia "*swich*". Los valores de la variable deben estar posterior a la sentencia "*case*" seguida de "**:**". Posterior a cada "*case*" se implementan las instrucciones del proceso correspondiente. Al finalizar las instrucciones se debe colocar la sentencia "*break*", con el fin de salir del proceso. En caso de que no se coloque la sentencia "*break*" en un proceso, se ejecutará el proceso del siguiente "*case*" hasta que se encuentre la sentencia "*break*". Opcionalmente, el último caso puede contener la sentencia "*default*", la cual ejecutaría el correspondiente conjunto de instrucciones si ninguno de los anteriores es verdadero. Por otro lado, la variable de la sentencia "*switch*" solo puede ser de tipo *int* o tipo *char*. La sintaxis de esta sentencia es:

```
switch(n){
   case 1:
       instrucción 1.1;
       instrucción 1.2;
       ..
       instrucción 1.n;
       break;
   case valor 2:
       instrucción 2.1;
       instrucción 2.2;
       ..
       instrucción 2.n;
       break;
   default:
       instrucción 3.1;
       instrucción 3.2;
       ..
       instrucción 3.n;
       break;
}
```

Ejemplo:

```
switch(variable){
   case 1:
```

```
        System.out.println("El valor de n es uno");
        break;
    case 2:
        System.out.println("El valor de n es dos");
        break;
    case 3:
        System.out.println("El valor de n es tres");
        break;
    default:
        System.out.println("El valor de n es mayor que tres");
        break;
}
```

2.3.7 Estructura de repetición *while*

La estructura de repetición "*while*" define un proceso iterativo, es decir, un proceso que se repetirá mientras que una condición tenga el valor verdadero. La sintaxis de esta sentencia es:

```
while(condición) {
    instrucción 1;
    instrucción 2;
    ..
    instrucción n;
}
```

Ejemplo:

```
int i=1;
while(i<=5) {
    System.out.println(i);  //imprime los números del 1 al 5.
    i++;
}
```

Nota: es posible abortar el proceso iterativo mediante la sentencia "*break*";

Ejemplo:

```
int i=1;
while(i>0) {
    System.out.println(i);
    if(i==10){
        break;
    }
    i++;
}
```

2.3.8 Estructura de repetición *for*

La estructura de repetición "*for*" define un proceso iterativo. La estructura "*for*" se compone de inicialización, condición e incremento. La inicialización debe incluir al menos una variable y un valor, la condición debe involucrar la variable de la inicialización con el fin de tener un mecanismo de terminar el proceso iterativo y el incremento debe también, involucrar la variable de la inicialización. En la inicialización es posible declarar la variable correspondiente. La sintaxis de esta sentencia es:

```
for(inicialización; condición; incremento) {
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
}
```

Ejemplo:

```
for(i=1; i<=5; i++) {
   System.out.println(i);  //imprime los números del 1 al 5.
}
```

Ejemplo con declaración:

```
for(int i=1; i<=5; i++) {
   System.out.println(i);  //imprime los números del 1 al 5.
}
```

Nota: es posible abortar el proceso iterativo mediante la sentencia "*break*";

2.3.9 Estructura de repetición *do while*

La estructura de repetición "*do*" "*while*" define un proceso iterativo. Contiene una diferencia con respecto a la estructura "*while*" que consiste en que el "*do*" "*while*" primero ejecuta y luego consulta, mientras que el "*while*" primero consulta y luego ejecuta. La sintaxis de esta sentencia es:

```
do{
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
}while(condición);
```

Ejemplo:

```
int i=1
do {
   System.out.println(i);  //imprime los números del 1 al 5.
   i++;
}while(i<=5);
```

Nota: es posible abortar el proceso iterativo mediante la sentencia "*break*";

2.4 Secuencias de escape

Una secuencia de escape es una sentencia que sirve para representar caracteres especiales. Estos caracteres inician con el símbolo "\" lo que indica que, a continuación se representa un carácter especial.

Tabla **11.** Secuencias de escape

Secuencia	Descripción
\b	*BackSpace*
\t	Tabulación
\n	Salto de línea
\"	Comilla doble
\'	Comilla sencilla
\\	*BackSlash*

2.5 Ejercicios propuestos

1. Un número par es aquel que es divisible en 2 y un número impar es aquel que no es divisible en 2. Con base en la anterior afirmación, implemente un algoritmo que permita verificar si un número es par o impar.

2. El factorial de un número entero mayor a 0, equivale al producto de todos los números enteros desde 1 hasta el número al que se le desea calcular el factorial. Esto significa que n! = 1 x 2 x 3 x ... x (n-1) x n. Además, el factorial de 0 es 1. Implemente un algoritmo que calcule el factorial de un número n.

3. La serie de Fibonacci consiste en una serie de números enteros positivos que inicia con los números 0 y 1. El número siguiente corresponde a la suma de los 2 anteriores. Entonces la serie de Fibonacci es: 0, 1, 1, 2, 3, 5, 8, 13, 21,… implemente un algoritmo que imprima la serie de Fibonacci.

4. La potenciación se resuelve elevando un número conocido como base a un número conocido como exponente. La potenciación indica que una base elevada a un exponente es igual a la sumatoria de la base, el número de veces que indica el exponente. Por ejemplo, 5^3=5+5+5. Implemente un algoritmo que resuelva la potenciación.

5. Un número primo es aquel que es divisible solo por 1 y por sí mismo. Para calcular si un número es primo es necesario verificar que el número a calcular no sea divisible por todos los números menores a él. Implemente un algoritmo que calcule si un número es primo.

CAPÍTULO 3

Conceptos básicos de programación orientada a objetos

La programación orientada a objetos se define como un paradigma que permite realizar una abstracción de la realidad, que se puede implementar en una aplicación de *software* con el fin de resolver problemas mediante el uso de un lenguaje de programación.

El paradigma de orientación a objetos comprende una gran cantidad de conceptos que permite el desarrollo de aplicaciones robustas.

3.1 Paquete

Un paquete es un contenedor de clases. Se utiliza para ordenar el código de forma consistente de acuerdo a los servicios implementados. Para que un código se encuentre contenido en un paquete es necesario agregar la siguiente sentencia.

```
package MiPaquete;
```

En donde “*Mi Paquete*” es el nombre del paquete que contendrá el código. Por otro lado, si se desea hacer uso de servicios implementados en otros paquetes se debe agregar el siguiente código.

```
import OtroPaquete;
```

Java contiene una gran cantidad de paquetes que proveen una gran cantidad de servicios. Algunos de estos paquetes se presentan en la Tabla 12.

Tabla 12. Paquetes básicos del *API* de Java

Paquete	Descripción
java.applet	Provee clases necesarias para crear *applets*.
java.awt	Contiene todas las clases para crear interfaces gráficas para pintar gráficas e imágenes.
java.awt.color	Provee clases para definiciones de color.
java.awt.event	Provee interfaces y clases para manejar eventos de componentes gráficos.
java.awt.font	Provee interfaces y clases relacionadas con fuentes.
java.awt.geom	Provee clases 2D para definir operaciones relacionadas con geometría de dos dimensiones.
java.awt.image	Provee interfaces y clases para crear y modificar imágenes.
java.awt.print	Provee interfaces y clases para usar el *API* de impresión.
java.beans	Provee interfaces y clases para el desarrollo de *beans,* que hace referencia a componentes basados en *JavaBeans™ architecture*.
java.beans.beancontext	Provee interfaces y clases relacionadas con *bean context*.
java.io	Provee interfaces y clases para entrada y salida de datos serializables.

java.lang	Provee clases fundamentales para el diseño del lenguaje de programación Java.
java.math	Provee clases para optimizar la precisión de entero aritmético (*BigInteger*) decimal aritmético (*BigDecimal*).
java.net	Provee clases para implementar aplicaciones de red.
java.rmi	Provee interfaces y clases para servicios *RMI*.
java.security	Provee interfaces y clases para el *framework* de seguridad.
java.sql	Provee interfaces y clases para procesar datos almacenados en fuentes de datos como bases de datos.
java.text	Provee interfaces y clases para manipular texto, fechas, números y mensajes.
java.util	Contiene el *framework* de colecciones, modelo de eventos, servicios de fecha y tiempo, internacionalización y clases misceláneas.

3.2 Clase

Una clase se define como un tipo abstracto de dato que contiene atributos y métodos. A través de una clase se implementa un concepto abstraído de la realidad. En este caso, los atributos hacen referencia a las características del concepto abstraído y los métodos hacen referencia a los servicios de dicho concepto.

La sintaxis de la clase debe ser la siguiente:

```
public class MiClase{
   //Definición de atributos
   //Definición de métodos
}
```

En Java se debe seguir una buena práctica que consiste en implementar cada clase en un archivo independiente con extensión ***.java***. Para el ejemplo anterior, el archivo debe denominarse *MiClase.java*.

3.2.1 Atributos

Los atributos hacen referencia a las características que se le incluyen a la clase. Estos atributos pueden ser declaraciones de tipos primitivos de datos o declaraciones de clases.

3.2.2 Visibilidad

La visibilidad se refiere al nivel de accesibilidad de los atributos y métodos. Los niveles de accesibilidad se dan por los siguientes términos:

1. *private.* Se puede acceder desde un método implementado desde la misma clase.
2. *public*. Se puede acceder desde un método implementado en cualquier clase.
3. *protected*. Se puede acceder desde un método implementado en una clase que herede la clase que contiene esta visibilidad y desde clases implementadas en el mismo paquete.

3.2.3 Métodos

Los métodos hacen referencia a los servicios que se le incluyen a la clase. En estos métodos se implementa el código necesario del servicio. Un método contiene los siguientes elementos:

1. Visibilidad. Se debe establecer si el método es *private*, *public* o *protected*.
2. Retorno. Un método puede retornar información. Si el método no retorna información se debe colocar la palabra reservada “*void*”.

El retorno puede ser un tipo primitivo de dato o una clase. Si un método tiene retorno, en la implementación del método, debe estar presente la palabra reservada "*return*".

3. Nombre. Identificador del método en la clase.
4. Parámetros. Un método puede recibir de *0* a *n* parámetros. Un parámetro puede ser un tipo primitivo de dato o una declaración de una clase. Los parámetros deben estar separados por comas.

Cada método implementa un código que debe estar contenido entre "*{*" y "*}*". La sintaxis de los métodos es la siguiente.

```
//método publico sin retorno y sin parámetros
public void miMetodo(){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
}

//método privado con retorno int y sin parámetros
prívate int miMetodo(){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
   return valorInt;
}

//método privado con retorno int y con parámetros
prívate int miMetodo(int parametro1, boolean parametro2, MiClase
parametro3){
   instrucción 1;
   instrucción 2;
   ..
   instrucción n;
   return valorInt;
}
```

3.2.4 Encapsulamiento

Es una característica que indica que los atributos que definen propiedades propias de la clase deben tener visibilidad private. De esta forma se ofrece seguridad a la información depositada en dichos atributos.

3.2.5 Apuntador *this*

El apuntador "*this*" permite acceder a los atributos y métodos de la clase. El uso del apuntador no es obligatorio, pero se recomienda usarlo como buena práctica. Es posible que el parámetro de un método tenga el mismo nombre que un atributo, en este caso el uso del apuntador *this* es obligatorio para que el compilador identifique si está haciendo referencia al atributo o al parámetro del método.

3.3 Objeto

Un objeto es la referencia e instancia de una clase. Al crear una referencia se asigna un espacio de memoria dinámica al objeto, pero no es utilizable. Al crear la instancia, el objeto es utilizable. La sintaxis de la referencia es la siguiente.

```
MiClase m;
```

Donde *m* es la referencia del objeto. La sintaxis de la instancia es:

```
m = new MiClase();
```

Al hacer la instancia se puede acceder a los atributos y métodos públicos y protegidos si aplica, a través del objeto *m*. Otra sintaxis para realizar referencia e instancia en la misma línea de código es:

```
MiClase m = new MiClase();
```

3.4 Sentencia *static*

Una clase puede tener atributos y/o métodos propios o no del objeto. La sentencia "*static*" define estos atributos y métodos de tal forma que puedan ser accedidos sin requerir una instancia de la clase. Por otro lado, un atributo "*static*" toma el mismo valor para todos los objetos que sean instancia de la clase que lo contiene. Por ejemplo, la clase *Math* contiene el método "*sin*" el cual calcula el seno de un parámetro dado.

Ejemplo:

```
public class MiClase{

   public static int miValor;

   public static long factorial(long n) {
       long fact=1;
       for(int i=1; i<n; i++){
          fact *= i;
       }
       return fact;
   }
}
```

En donde se puede hacer uso del método factorial de la siguiente forma.

```
long valor = MiClase.factorial(5);
```

También permite hacer uso del atributo *miValor* de la siguiente forma.

```
MiClase c1 = new MiClase();
c1.miValor = 10;
MiClase c2 = new MiClase();
MiClase c3 = new MiClase();
```

En el código anterior, el atributo *miValor* tendrá el valor 10 para los objetos c1, c2 y c3, solo con asignarlo en un objeto de ellos, que para el caso es en c1.

3.5 Sentencia final

Una clase puede tener atributos finales que hacen referencia a constantes que no pueden cambiar su valor en tiempo de ejecución de la aplicación. La sintaxis es la siguiente:

```
public class MiClase{

   public final static int uno=1;
   public final static int dos=2;

}
```

Por ejemplo el atributo "*PI*" cuyo valor se encuentra implementado en la clase del *API* de Java *Math,* puede ser accedido sin requerir instancia de la clase *Math* y su valor es constante.

3.6 Clasificación de métodos

Los métodos se pueden clasificar en cuatro tipos que son los siguientes:

1. Constructores. Un constructor es el primer método que se ejecuta al realizar la instancia de un objeto. Uno de los usos principales de un constructor es la inicialización de los atributos de la clase. El método constructor debe tener visibilidad pública y no posee retorno. La sintaxis es la siguiente:

```
public class MiClase{

   //Definición de atributos
   prívate int atributo1;
   prívate int atributo2;

   //Definición de método constructor
   public MiClase(){
       this.atributo1=0;
       this.atributo1=0;
   }
}
```

2. Consultores. Un consultor es el método que permite retornar el valor de un atributo con visibilidad private al aplicar el concepto de encapsulamiento. La sintaxis es la siguiente:

```
public class MiClase{

   //Definición de atributos
   prívate int atributo1;
   prívate int atributo2;

   //Método constructor
   public MiClase(){
       this.atributo1=0;
       this.atributo1=0;
   }
```

```
    //Método consultor
    public int getAtributo1() {
        return this.atributo1;
    }

    //Método consultor
    public int getAtributo2() {
        return this.atributo2;
    }
}
```

3. Modificadores. Un modificador es el método que permite asignar valor a un atributo con visibilidad *private* al aplicar el concepto de encapsulamiento. La sintaxis es la siguiente:

```
public class MiClase{

    //Definición de atributos
    prívate int atributo1;
    prívate int atributo2;

    //Método constructor
    public MiClase(){
        this.atributo1=0;
        this.atributo1=0;
    }

    //Método consultor
    public int getAtributo1() {
        return this.atributo1;
    }

    //Método consultor
    public int getAtributo2() {
        return this.atributo2;
    }

    //Método modificador
    public void setAtributro1(int atributo1) {
        this.atributo1 = atributo1;
    }

    //Método modificador
    public void setAtributro2(int atributo2) {
        this.atributo2 = atributo2;
    }
}
```

4. Analizadores. Un analizador es el método que permite implementar la lógica del servicio del mismo, es decir, allí se implementan los algoritmos requeridos. La sintaxis es la siguiente:

```
public class MiClase{

   //Definición de atributos
   prívate int atributo1;
   prívate int atributo2;

   //Método constructor
   public MiClase(){
       this.atributo1=0;
       this.atributo1=0;
   }

   //Método consultor
   public int getAtributo1() {
       return this.atributo1;
   }

   //Método consultor
   public int getAtributo2() {
       return this.atributo2;
   }

   //Método modificador
   public void setAtributro1(int atributo1) {
       this.atributo1 = atributo1;
   }

   //Método modificador
   public void setAtributro2(int atributo2) {
       this.atributo2 = atributo2;
   }

   //Método analizador
   public int calcularMayor() {
       if(this.atributo1 > this.atributo2){
          return this.atributo1;
       }else{
          return this.atributo2;
       }
   }
}
```

3.7 Sobrecarga de métodos

La sobrecarga de métodos es una característica que permite que varios métodos en una misma clase tengan el mismo nombre. La forma en que el compilador identifica cuál es el método a utilizar en tiempo de ejecución, se debe a que estos deben poseer diferentes parámetros y/o retorno. La diferencia puede estar dada en el número de parámetros y/o en el tipo de los mismos. Por ejemplo se plantean los siguientes métodos sobrecargados.

```
//Método sobrecargado 1. Sin parámetro y sin retorno.
public void miMetodo(){

}

//Método sobrecargado 2. Con parámetro y con retorno int.
public int miMetodo(int parametro1){

}

//Método sobrecargado 3. Con parámetros y con retorno boolean.
public boolean miMetodo(int parametro1, int parametro2){

}

//Método sobrecargado 4. Con parámetros diferentes a la
sobrecarga 3 y con retorno boolean.
public boolean miMetodo(int parametro1, long parametro2){

}
```

3.8 Recursividad

La recursividad es la característica en la programación que permite hacer un llamado a un método desde el mismo método. Esta característica simplifica el desarrollo. Cada llamado recursivo equivale a una iteración en una estructura de repetición como el "*while*" o el "*for*". Tiene la ventaja de utilizar casi los mismos recursos que en un proceso iterativo regular. Por otro lado, existen algoritmos que necesariamente deben ser implementados de forma recursiva como algoritmos fractales y árboles.

Para aplicar el concepto de recursividad, el método debe necesariamente retornar un valor, recibir por parámetro al menos un valor, implementar una condición de ruptura del proceso recursivo e implementar una función recursiva.

Por ejemplo, si se desea implementar el algoritmo del factorial se podría implementar el siguiente método para resolver el algoritmo:

```
public long factorial(long n) {
   long fact=1;
   for(int i=1; i<n; i++){
       fact *= i;
   }
   return fact;
}
```

Entonces, suponiendo que *n*=5, el algoritmo realiza 5 iteraciones, en donde en cada iteración se presentan los siguientes resultados en las variables:

1. Primera iteración: fact=1*1=1
2. Segunda iteración: fact=1*2=2
3. Tercera iteración: fact=2*3=6
4. Cuarta iteración: fact=6*4=24
5. Quinta iteración: fact=24*5=120

También se puede hacer la implementación del algoritmo del factorial de forma recursiva.

```
public long factorial(long n) {
   if(n==1 || n==0){
       return 1;
   }else{
       return n*factorial(n-1);
   }
}
```

Otra forma más avanzada de codificar el mismo algoritmo recursivo es la siguiente:

```
public long factorial(long n) {
   return (n==1)?1:n*factorial(n-1);
}
```

Entonces, suponiendo que *n*=5, el algoritmo realiza 5 llamadas recursivas, en donde en cada llamada se presentan los siguientes resultados.

1. Primer llamada: retorna 5*factorial(4)
2. Segunda llamada: retorna 4*factorial(3)
3. Tercer llamada: retorna 3*factorial(2)
4. Cuarta llamada: retorna 2*factorial(1)
5. Quinta llamada: retorna 1

Para hacer el primer llamado al método se puede considerar la siguiente sentencia:

```
long f = factorial(5);
```

En la Figura 1 se observa cómo cada llamada aporta a la consecución del resultado del algoritmo.

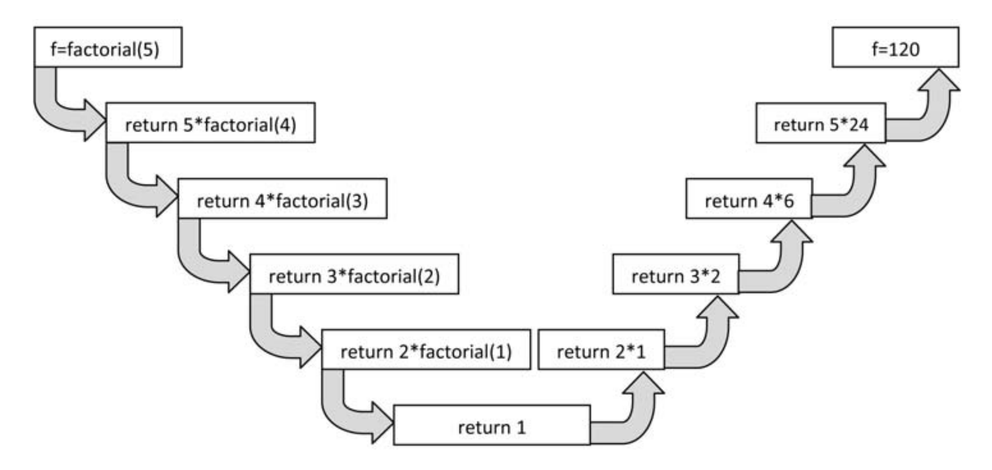


Figura 1. Representación de funcionamiento de recursividad

3.9 Bajo acoplamiento

Es la característica en el paradigma de orientación a objetos que indica que los diferentes subsistemas deben estar unidos de forma mínima. Esto indica, que las clases que se construyen deben ser lo

más reducidas, sin involucrar elementos que impliquen conceptos diferentes a los tratados en la clase.

Un ejemplo de bajo acoplamiento podría ser una memoria *USB*. Esta memoria es un sistema totalmente independiente al sistema del computador. Se conectan los dos sistemas a través del puerto *USB* y de forma automática, ambos sistemas quedan integrados. Además, esta operación se puede realizar mientras el computador se encuentre en funcionamiento. Así mismo es posible desconectar la memoria sin problemas.

Un ejemplo de alto acoplamiento, con base en el anterior es el disco duro. Este se encuentra altamente acoplado, debido a que no se puede desconectar mientras el computador se encuentra en funcionamiento. Además, sin este sistema el computador no puede funcionar. Por otro lado, un ejemplo de bajo acoplamiento es la memoria *USB*, la cual se puede conectar y desconectar de forma simple y no afecta ni al computador ni a la memoria misma.

3.10 Alta cohesión

Es la característica en el paradigma de orientación a objetos que indica que, las propiedades y servicios de una clase deben ser consistentes con el concepto que abstrae dicha clase.

Por ejemplo, si se tiene una clase *Triángulo*, esta clase podría contener los siguientes atributos:

- Identificación
- Base
- Altura

Además podría tener servicios como:

- Calcular Área
- Calcular Perímetro

En este ejemplo la clase *Triángulo* se encuentra altamente cohesionada ya que los atributos y métodos hacen referencia

directa a características y comportamientos de concepto abstraído que es el triángulo.

Si se incluye por ejemplo el método calcular volumen, esta clase estaría bajamente cohesionada, ya que un triángulo no posee volumen. Este método tendría que ser trasladado a la clase pirámide.

3.11 Manejo de excepciones

En el lenguaje Java, una "*Exception*" hace referencia a una condición anormal que se produce en tiempo de ejecución de la aplicación. Algunas excepciones son denominadas fatales, las cuales provocan la finalización de la ejecución de la aplicación. Generalmente, las excepciones se generan por que falla la operación como consecuencia de un error de uso de la aplicación por parte del usuario. Para ilustrar el concepto, se presentan los siguientes ejemplos:

- Si el usuario intenta abrir un archivo e ingresa de forma incorrecta la ruta del mismo, la aplicación presenta una excepción que debe controlarse para presentarle información de error de ruta del archivo al usuario.

- Si el usuario desea ingresar un número para realizar una operación aritmética, pero erróneamente ingresa un carácter, la aplicación presenta una excepción de formato de número que debe controlarse para indicarle al usuario que no se puede realizar la operación aritmética.

Las excepciones se representan mediante clases derivadas de la clase *Throwable*, sin embargo, las clases con las que se desarrolla, se derivan de la clase *Exception* que pertenece al paquete **java.lang**.

3.11.1 Estructura *try*, *catch* y *finally*

Las excepciones en Java deben ser capturadas mediante el uso de las estructuras "*try*", "*catch*" y "*finally*". En el bloque *try* se debe implementar el código del proceso que se desea ejecutar. En el

bloque *catch* se implementa el código alterno que se ejecutará en caso de que se presente una situación anormal o excepción en la ejecución del código implementado en el bloque *try*. Es posible tener varios bloques *catch* que resuelvan diferentes tipos de excepción. El bloque *finally* es opcional, pero en caso de implementarse, este se ejecutará independientemente, si se presenta o no excepción. Este se implementa posterior a la implementación del bloque *try* y del bloque *catch*. La sintaxis es la siguiente:

```
public void miMetodo(){
   ..
   try{
       instrucción 1;
       instrucción 2;
       ..
       instrucción n;
   }catch(Exception e){
       //Instrucciones del manejo de la excepcion
   }finally{
       //Instrucciones que se ejecutan en cualquiera de los dos casos
   }
   ..
}
```

3.11.2 Sentencia *throws*

En caso que el código de un método genere una "*Exception*", pero no se desee manejar dicha excepción, es posible enviar el manejo de la misma al método que hace el llamado. Este envío del manejo de la excepción se realiza mediante la inclusión de la sentencia "*throws*" seguida del nombre de la excepción posterior a los parámetros del método. Esta sentencia obliga a que el método que hace el llamado, implemente el manejo de la excepción a través del bloque "*try catch*" o envíe a su vez la excepción al método que hace el llamado a través de la sentencia "*throws*". La sintaxis es la siguiente.

```
public void miMetodo()throws Exception{
   ..
}
```

3.11.3 Excepciones estándar del *API* de Java

Las excepciones en Java se representan mediante dos tipos de clases derivadas de la clase *Throwable* que son *Error* y *Exception*.

La clase *Error* está relacionada con errores de compilación, errores del sistema o errores de la *JVM*. Estos errores son irrecuperables y no dependen del desarrollador.

La clase *Exception* es la que debe tener en cuenta el desarrollador, debido a que de esta se derivan clases que manejan las excepciones que pueden ser controladas en tiempo de ejecución. Las clases derivadas de *Exception* más usuales son:

1. **RuntimeException***:* contiene excepciones frecuentes en tiempo de ejecución de la aplicación.
2. **IOException***:* contiene excepciones relacionadas con entrada y salida de datos.

Las clases derivadas de *Exception* pueden pertenecer a distintos paquetes de Java. Algunas de ellas pertenecen a **java.lang**, otras a **java.io** y a otros paquetes. Por derivarse de la clase *Throwable* todos los tipos de excepciones pueden usar los métodos siguientes:

1. *String getMessage():* extrae el mensaje asociado con la excepción.
2. *String toString():* devuelve un String que describe la excepción.
3. *void printStackTrace():* indica el método donde se lanzó la excepción.

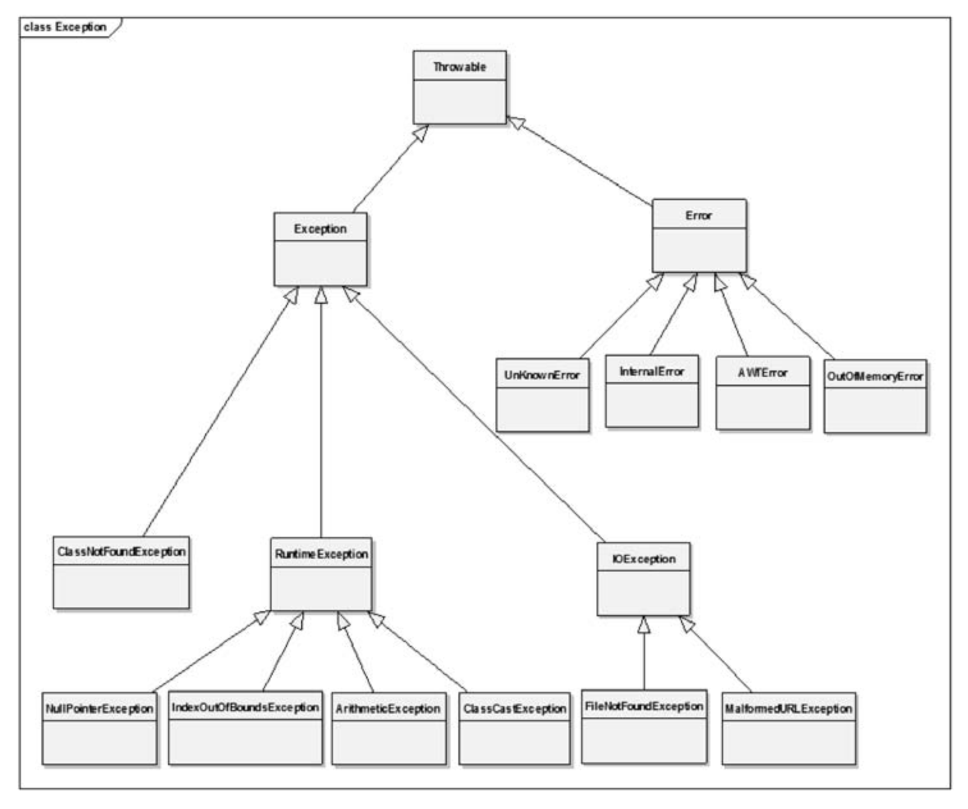


Figura 2. Jerarquía simplificada de clases derivadas de *Throwable*

3.11.4 Creación de excepciones en Java

En un proyecto es posible crear excepciones propias solo con heredar de la clase *Exception* o de una de sus clases derivadas. Las clases derivadas de *Exception* suelen tener dos constructores:

1. Un constructor sin argumentos.
2. Un constructor que recibe un *String* como argumento. En este *String* se suele definir un mensaje que explica el tipo de excepción generada. Este mensaje debe enviarse a la clase *Exception* mediante la sentencia *super(String).*

La sintaxis es la siguiente:

```
class MiExcepcion extends Exception {
   public MiExcepcion() { // Constructor por defecto
       super();
   }
   public MiExcepción(String s) { // Constructor con mensaje
       super(s);
   }
}
```

3.12 Ejercicios propuestos

1. Implemente una clase denominada *Cuadrado* que contenga un atributo privado, dos métodos constructores sobrecargados con y sin parámetros, métodos consultores, métodos modificadores y métodos analizadores que calculen el área y perímetro del cuadrado.
2. Implemente una clase denominada *Triángulo* que contenga un atributo privado, dos métodos constructores sobrecargados con y sin parámetros, métodos consultores, métodos modificadores y métodos analizadores que calculen el área y perímetro del triángulo.
3. Implemente una clase denominada *Rectángulo* que contenga un atributo privado, dos métodos constructores sobrecargados con y sin parámetros, métodos consultores, métodos modificadores y métodos analizadores que calculen el área y perímetro del rectángulo.
4. Implemente una clase denominada *Operaciones*, que contenga métodos estáticos que calculen el factorial de un número y que verifique si un número es primo.

CAPÍTULO 4

Clases de utilidad en Java

Dentro del *API* de Java existe una gran colección de clases que son muy utilizadas en el desarrollo de aplicaciones. Entre las clases de utilidad de Java más utilizadas y conocidas están las siguientes: *String, Integer, Double, Float, Long, Boolean, Math, Date, StringTokenizer* y*BigInteger*.

4.1 Clase *String*

La clase *String* está orientada al manejo de cadenas de caracteres y pertenece al paquete **java.lang** del *API* de Java. Los objetos que son instancia de la clase *String*, se pueden crear a partir de cadenas constantes también llamadas literales, las cuales deben estar contenidas entre comillas dobles. En la clase *String*, se puede asignar cadenas de las dos formas siguientes:

```
String cadena1 = new String(“Hola”);   //Creación a través de constructor
String cadena2 = “Hola”;               //Creación a través de literal
```

El método de creación a través de literal es el más eficiente, porque al encontrar un texto entre comillas se crea automáticamente un objeto de la clase *String*.

Tabla 13. Métodos principales de la clase *String*

Retorno	Método	Descripción
void	*String()*	Constructor que inicializa un objeto con una secuencia de caracteres vacía.
void	*String(char[] value)*	Constructor que crea un *String* inicializa un objeto con una secuencia de caracteres tipo *char.*
char	*charAt(int index)*	Retorna el carácter especificado en la posición *index.*
int	*compareTo(String anotherString)*	Compara dos cadenas de caracteres alfabéticamente. Retorna 0 si son iguales, entero negativo si la primera es menor o entero positivo si la primera es mayor.
String	*concat(String str)*	Concatena la cadena del parámetro al final de la primera cadena.
boolean	*contains(CharSequence s)*	Retorna *true* si la cadena contiene la secuencia tipo *char* del parámetro.
boolean	*endsWith(String suffix)*	Retorna verdadero si el final de la cadena es igual al sufijo del parámetro.
boolean	*equals(Object anObject)*	Retorna verdadero si la cadena es igual al objeto del parámetro.
int	*indexOf(String str)*	Retorna el índice de la primera ocurrencia de la cadena del parámetro.
boolean	*isEmpty()*	Retorna verdadero si la longitud de la cadena es 0.
int	*length()*	Retorna la longitud de la cadena.

String	*replace(char oldChar, char newChar)*	Retorna una nueva cadena reemplazando los caracteres del primer parámetro con el carácter del segundo parámetro.
String[]	*split(String regex)*	Retorna un conjunto de cadenas separadas por la cadena del parámetro.
boolean	*startsWith(String prefix)*	Retorna verdadero si el comienzo de la cadena es igual al prefijo del parámetro.
String	*substring(int beginIndex)*	Retorna la subcadena desde el carácter del parámetro.
String	*substring(int beginIndex, int endIndex)*	Retorna la subcadena desde el carácter del primer parámetro hasta el carácter del segundo parámetro.
char[]	*toCharArray()*	Retorna el conjunto de caracteres de la cadena.
String	*toLowerCase()*	Retorna la cadena en minúsculas.
String	*toUpperCase()*	Retorna la cadena en mayúsculas.
static String	*valueOf(char[] data)*	Convierte en cadena el conjunto de caracteres del parámetro.
static String	*valueOf(double d)*	Convierte en cadena el dato del parámetro.
static String	*valueOf(float f)*	Convierte en cadena el dato del parámetro.
static String	*valueOf(int i)*	Convierte en cadena el dato del parámetro.
static String	*valueOf(long l)*	Convierte en cadena el dato del parámetro.
static String	*valueOf(Object obj)*	Convierte en cadena el objeto del parámetro.

Ejemplo de uso del método *length* y *charAt*

```
public class EjemplosString {

   public static void main(String[] args) {
       String cadena="Hola Mundo";
       char caracter;
       System.out.println("La cadena tiene "+cadena.length()+
       "caracteres");
       for(int i=0; i<cadena.length(); i++){
          caracter=cadena.charAt(i);
          System.out.println("El caracter en la posición "+
          i+" es: "+caracter);
       }
   }
}
```

Salida estándar

```
La cadena tiene 10 caracteres
El caracter en la posición 0 es: H
El caracter en la posición 1 es: o
El caracter en la posición 2 es: l
El caracter en la posición 3 es: a
El caracter en la posición 4 es:
El caracter en la posición 5 es: M
El caracter en la posición 6 es: u
El caracter en la posición 7 es: n
El caracter en la posición 8 es: d
El caracter en la posición 9 es: o
```

Ejemplo de uso del método *concat*

```
public class EjemplosString {

   public static void main(String[] args) {
       String cadena1="Hola";
       String cadena2="Mundo";
       System.out.println("La cadena 1 es: "+cadena1);
       System.out.println("La cadena 2 es: "+cadena2);
       System.out.println("El texto concatenado es: "+
       cadena1.concat(cadena2));
   }
}
```

Salida estándar

```
La cadena 1 es: Hola
La cadena 2 es: Mundo
El texto concatenado es: HolaMundo
```

Ejemplo de uso del método *replace*

```
public class EjemplosString {

   public static void main(String[] args) {
       String cadena="Este es el texto original";
       System.out.println("La cadena es: "+cadena);
       System.out.println("La cadena modificada es: "+
       cadena.replace('e', '?'));
   }
}
```

Salida estándar
`La cadena es: Este es el texto original` `La cadena modificada es: Est? ?s ?l t?xto original`

Ejemplo de uso del método *split*

```
public class EjemplosString {

   public static void main(String[] args) {
       String cadena="Hola planeta tierra";
       String cadenas[]=cadena.split(" ");
       for(int i=0; i<cadenas.length; i++){
          System.out.println("La cadena "+i+" es: "+cadenas[i]);
       }
   }
}
```

Salida estándar

`La cadena 0 es: Hola`
`La cadena 1 es: planeta`
`La cadena 2 es: tierra`

Ejemplo de uso del método *substring*

```
public class EjemplosString {

   public static void main(String[] args) {
       String cadena="Hola planeta tierra";
       System.out.println("La cadena es: "+cadena);
       System.out.println("La sub cadena del caracter 2 al 10 es:
       "+cadena.substring(2, 10));
   }
}
```

Salida estándar
`La cadena es: Hola planeta tierra` `La subcadena del caracter 2 al 10 es: la plane`

Ejemplo de uso de los métodos *toUpperCase* y *toLowerCase*

```
public class EjemplosString {

   public static void main(String[] args) {
       String cadena="Hola Mundo";
       System.out.println("El texto normal es: "+cadena);
       System.out.println("El texto en mayúsculas es: "+
       cadena.toUpperCase());
       System.out.println("El texto en minúsculas es: "+
       cadena.toLowerCase());
   }
}
```

Salida estándar

`El texto normal es: Hola Mundo`
`El texto en mayúsculas es: HOLA MUNDO`
`El texto en minúsculas es: hola mundo`

4.2 Clase *Integer*

La clase *Integer* permite convertir un tipo primitivo de dato *int* a objeto *Integer*. La clase *Integer* pertenece al paquete **java.lang** del *API* de Java y hereda de la clase *java.lang.Number*. Provee métodos para realizar diferentes tipos de conversiones relacionados con el tipo primitivo de dato *int*.

Tabla 14. Métodos principales de la clase *Integer*

Retorno	Método	Descripción
void	*Integer(int value)*	Constructor que inicializa un objeto con un dato primitivo.
void	*Integer(String s)*	Constructor que inicializa un objeto con una cadena de caracteres. Esta cadena debe contener un número entero.

int	*compareTo(Integer anotherInteger)*	Compara dos objetos *Integer* numéricamente.
double	*doubleValue()*	Retorna el valor del *Integer* en tipo primitivo *double.*
boolean	*equals(Object obj)*	Compara el *Integer* con el objeto del parámetro.
float	*floatValue()*	Retorna el valor del *Integer* en tipo primitivo *float.*
int	*intValue()*	Retorna el valor del *Integer* en tipo primitivo *int.*
long	*longValue()*	Retorna el valor del *Integer* en tipo primitivo *long.*
static int	*parseInt(String s)*	Convierte la cadena de caracteres del parámetro en tipo primitivo *int.*
short	*shortValue()*	Retorna el valor del *Integer* en tipo primitivo *short*.
static String	*toBinaryString(int i)*	Retorna el número del parámetro en su correspondiente cantidad binaria en una cadena de caracteres.
static String	*toHexString(int i)*	Retorna el número del parámetro en su correspondiente cantidad hexadecimal en una cadena de caracteres.
static String	*toOctalString(int i)*	Retorna el número del parámetro en su correspondiente cantidad octal en una cadena de caracteres.
String	*toString()*	Retorna el valor del *Integer* en una cadena de caracteres.
static Integer	*valueOf(int i)*	Retorna el número del parámetro en un objeto *Integer*.
static Integer	*valueOf(String s)*	Retorna la cadena del parámetro en un objeto *Integer*.

Nota: las clases *Byte*, *Short, Long, Double* y *Float* tienen las mismas características de la clase *Integer*, diferenciándose por el tamaño del dato en el caso de *Byte*, *Short* y *Long;* y por la capacidad de número real en el caso de *Double* y *Float*.

Ejemplo de uso de los métodos *intValue* y *floatValue*

```
public class EjemplosInteger {

   public static void main(String[] args) {
       Integer dato=new Integer("10");
       System.out.println("El int es: "+dato.intValue());
       System.out.println("El float es: "+dato.floatValue());

   }
}
```

Salida estándar

```
El int es: 10
El float es: 10.0
```

Ejemplo de uso de los métodos *toBinaryString*, *toOctalString* y *toHexString*

```
public class EjemplosInteger {

   public static void main(String[] args) {
       int decimal=1000;
       String binario=Integer.toBinaryString(decimal);
       String octal=Integer.toOctalString(decimal);
       String hexa=Integer.toHexString(decimal);
       System.out.println("El número decimal es: "+decimal);
       System.out.println("El número binario es: "+binario);
       System.out.println("El número octal es: "+octal);
       System.out.println("El número hexadecimal es: "+hexa);
   }
}
```

Salida estándar

```
El número decimal es: 1000
El número binario es: 1111101000
El número octal es: 1750
El número hexadecimal es: 3e8
```

4.3 Clase *Boolean*

La clase *Boolean* permite convertir un tipo primitivo de dato boolean a objeto boolean. La clase booleana pertenece al paquete **java.lang** del *API* de Java.

Provee métodos para realizar diferentes tipos de conversiones relacionados con el tipo primitivo de dato booleano.

Tabla 15. Métodos principales de la clase *Boolean*

Retorno	Método	Descripción
void	*Boolean(boolean value)*	Constructor que inicializa un objeto con un dato primitivo.
void	*Boolean(String s)*	Constructor que inicializa un objeto con una cadena de caracteres. Si la cadena y es igual al texto "*true*" el valor es verdadero.
boolean	*booleanValue()*	Retorna el valor del *Boolean* en tipo primitivo boolean.
int	*compareTo(Boolean b)*	Compara el valor del *Boolean* con el valor del parámetro.
static boolean	*parseBoolean(String s)*	Convierte la cadena de caracteres del parámetro en tipo primitivo boolean.
String	*toString()*	Retorna el valor del *Boolean* en una cadena de caracteres.
static Boolean	*valueOf(boolean b)*	Retorna el valor booleano del parámetro en un objeto booleano.
static Boolean	*valueOf(boolean b)*	Retorna la cadena del parámetro en un objeto booleano.

4.4 Clase *Math*

La clase *Math* contiene métodos para utilizar operaciones numéricas básicas y funciones trigonométricas. La clase *Math* pertenece al paquete **java.lang** del *API* de Java. La clase *Math* se define como "*final*" que indica que no puede tener instancia, de tal forma que sus atributos y métodos son "*static*".

Tabla 16. Atributos de la clase *Math*

Tipo	Atributo	Descripción
static double	E	Retorna la base de logaritmo natural que es 2.718281828459045
static double	PI	Retorna el número *pi* que es equivalente al perímetro de la circunferencia dividido el radio. Su valor es 3.141592653589793

Tabla 17. Métodos principales de la clase *Math*

Retorno	Método	Descripción
static double	*abs(double a)*	Retorna el número absoluto del parámetro *double.*
static float	*abs(float a)*	Retorna el número absoluto del parámetro *float*.
static int	*abs(int a)*	Retorna el número absoluto del parámetro *int*.
static long	*abs(long a)*	Retorna el número absoluto del parámetro *long*.
static double	*acos(double a)*	Retorna el arco coseno del valor del parámetro. El ángulo retornado está entre 0.0 y *pi*.
static double	*asin(double a)*	Retorna el arco seno del valor del parámetro. El ángulo retornado está entre –*pi*/2 y *pi*/2.

static double	*atan(double a)*	Retorna el arco tangente del valor del parámetro. El ángulo retornado está entre –*pi*/2 y *pi*/2.
static double	*cbrt(double a)*	Retorna la raíz cubica del valor del parámetro.
static double	*cos(double a)*	Retorna el coseno del ángulo del parámetro.
static double	*cosh(double x)*	Retorna el coseno hiperbólico del ángulo del parámetro.
static double	*exp(double a)*	Retorna el número de Eurler basado de la potencia del parámetro.
static double	*log(double a)*	Retorna el logaritmo natural del valor del parámetro.
static double	*log10(double a)*	Retorna el logaritmo base 10 del valor del parámetro.
static double	*max(double a, double b)*	Retorna el número mayor entre los parámetros *double*.
static float	*max(float a, float b)*	Retorna el número mayor entre los parámetros *float*.
static int	*max(int a, int b)*	Retorna el número mayor entre los parámetros *int*.
static long	*max(long a, long b)*	Retorna el número mayor entre los parámetros *long*.
static double	*min(double a, double b)*	Retorna el número menor entre los parámetros *double*.
static float	*min(float a, float b)*	Retorna el número menor entre los parámetros *float*.
static int	*min(int a, int b)*	Retorna el número menor entre los parámetros *int*.

static long	*min(long a, long b)*	Retorna el número menor entre los parámetros *long*.
static double	*pow(double a, double b)*	Retorna la potencia del primer parámetro elevado al segundo parámetro.
static double	*random()*	Retorna un número *double* aleatorio entre 0.0 y 1.0.
static long	*round(double a)*	Retorna el número *long* más cercano del parámetro *double*.
static int	*round(float a)*	Retorna el número *int* más cercano del parámetro *float*.
static double	*sin(double a)*	Retorna el seno del ángulo del parámetro.
static double	*sinh(double a)*	Retorna el seno hiperbólico del ángulo del parámetro.
static double	*sqrt(double a)*	Retorna la raíz cuadrada del valor del parámetro.
static double	*tan(double a)*	Retorna la tangente del ángulo del parámetro.
static double	*tanh(double a)*	Retorna la tangente hiperbólica del ángulo del parámetro.
static double	*toDegrees(double angrad)*	Convierte un ángulo de radianes a grados.
static double	*toRadians (double angdeg)*	Convierte un ángulo de grados a radianes.

Ejemplo de uso de los métodos *cbrt*, *sqrt* y *pow*

```
public class EjemplosMath {

   public static void main(String[] args) {
       double dato=144;
       double raizCuadrada = Math.sqrt(dato);
       System.out.println("La raíz cuadrada de: "+dato+
       " es: "+raizCuadrada);
       double raizCubica = Math.cbrt(dato);
       System.out.println("La raíz cúbica de: "+dato+
       " es: "+raizCubica);
       double base=5;
       double exponente=4;
       double potencia=Math.pow(base, exponente);
       System.out.println("La potencia de "+base+
       " ^ "+exponente+" es: "+potencia);
   }
}
```

Salida estándar
La raíz cuadrada de: 144.0 es: 12.0 La raíz cúbica de: 144.0 es: 5.241482788417793 La potencia de 5.0 ^ 4.0 es: 625.0

Ejemplo de uso del atributo *PI* y de los métodos *sin*, *cos*, *round* y *toRadians*

```
public class EjemplosMath{

   public static void main(String[] args) {
       double seno;
       double coseno;
       for(int i=0;i<=360;i+=90){
          seno=Math.round(Math.sin(i*Math.PI/180));
          System.out.println("El seno de "+i+" usando PI/180 es: "+
          seno);
          seno=Math.round(Math.sin(Math.toRadians(i)));
          System.out.println("El seno de "+i+" usando toRadians es: "+
          seno);
          coseno=Math.round(Math.cos(i*Math.PI/180));
          System.out.println("El coseno de "+i+" usando PI/180 es: "+
          coseno);
          coseno=Math.round(Math.cos(Math.toRadians(i)));
          System.out.println("El coseno de "+i+" usando toRadians es: "+
          coseno);
       }
   }
}
```

Salida estándar

```
El seno de 0 usando PI/180 es: 0.0
El seno de 0 usando toRadians es: 0.0
El coseno de 0 usando PI/180 es: 1.0
El coseno de 0 usando toRadians es: 1.0
El seno de 90 usando PI/180 es: 1.0
El seno de 90 usando toRadians es: 1.0
El coseno de 90 usando PI/180 es: 0.0
El coseno de 90 usando toRadians es: 0.0
El seno de 180 usando PI/180 es: 0.0
El seno de 180 usando toRadians es: 0.0
El coseno de 180 usando PI/180 es: -1.0
El coseno de 180 usando toRadians es: -1.0
El seno de 270 usando PI/180 es: -1.0
El seno de 270 usando toRadians es: -1.0
El coseno de 270 usando PI/180 es: 0.0
El coseno de 270 usando toRadians es: 0.0
El seno de 360 usando PI/180 es: 0.0
El seno de 360 usando toRadians es: 0.0
El coseno de 360 usando PI/180 es: 1.0
El coseno de 360 usando toRadians es: 1.0
```

4.5 Clase *Date*

La clase *Date* representa un instante de tiempo especifico con una precisión en milisegundos. Adicionalmente, la clase *Date* permite el uso del formato *Universal Coordinated Time, UTC*. Por otro lado, muchos computadores están definidos en términos de *Greenwich Mean Time, GMT,* que es equivalente a *Universal Time, UT*. *GMT* es el nombre estándar y *UT* es el nombre científico del estándar. La diferencia entre *UT* y *UTC* es que, *UTC* está basado en un reloj atómico y *UT* está basado en un reloj astronómico.

Las fechas en Java comienzan en el valor "*standar based time*" llamado "*epoch*" que hace referencia al 1 de enero de 1970, 0 horas 0 minutos 0 segundos *GMT*.

La clase *Date* posee métodos que permiten la manipulación de fechas. La clase *Date* pertenece al paquete **java.util** del *API* de Java.

Tabla 18. Métodos principales de la clase *Date*

Retorno	Método	Descripción
void	*Date()*	Constructor que inicializa la fecha en el milisegundo más cercano a la fecha del sistema.
void	*Date(long date)*	Constructor que inicializa la fecha en milisegundos del parámetro a partir del "*epoch*".
boolean	*after(Date when)*	Retorna verdadero si la fecha esta después de la fecha del parámetro.
boolean	*before(Date when)*	Retorna verdadero si la fecha esta antes de la fecha del parámetro.
int	*compareTo(Date anotherDate)*	Compara la fecha con la del parámetro. Si retorna 0 las fechas son iguales.
boolean	*equals(Object obj)*	Retorna verdadero si la fecha es igual a la del objeto del parámetro.
long	*getTime()*	Retorna la fecha en milisegundos a partir del "*epoch*".
void	*setTime(long time)*	Asigna la fecha en milisegundos a partir del "*epoch*".
String	*toString()*	Retorna la fecha en una cadena de caracteres.

Ejemplo de uso del método *toString*

```
import java.util.Date;

public class EjemplosDate {

   public static void main(String[] args) {
       Date fecha=new Date();
       String fechaActual=fecha.toString();
       System.out.println("La fecha actual es: "+fechaActual);
   }
}
```

Salida estándar
`La fecha actual es: Thu Jul 01 02:11:54 GMT 2010`

4.6 Clase *StringTokenizer*

La clase *StringTokenizer* permite en una aplicación romper una cadena en unidades denominadas *token*. Una cadena se puede romper generando un *token* a través de un delimitador. Si este delimitador no se especifica, por defecto será un espacio. La clase *StringTokenizer* pertenece al paquete **java.lang** del *API* de Java.

Tabla 19. Métodos principales de la clase *StringTokenizer*

Retorno	Método	Descripción
void	*StringTokenizer(String str)*	Constructor que inicializa la cadena de caracteres con el parámetro. El delimitador por defecto será un espacio.
void	*StringTokenizer(String str, String delim)*	Constructor que inicializa la cadena de caracteres con el primer parámetro y el delimitador con el segundo parámetro.
void	*StringTokenizer(String str, String delim, boolean returnDelims)*	Constructor que inicializa la cadena de caracteres con el primer parámetro y el delimitador con el segundo parámetro. Si el tercer parámetro es *true*, cada *token* incluye el delimitador.
int	*countTokens()*	Retorna el número de *tokens* en la cadena.
boolean	*hasMoreTokens ()*	Retorna verdadero si hay más *tokens* disponibles en la cadena.
String	*nextToken()*	Retorna el siguiente *token* de la cadena.
String	*nextToken(String delim)*	Retorna el siguiente *token* de la cadena con base en el delimitador especificado en el parámetro.

Ejemplo de uso de los métodos *hasMoreTokens* y *nextToken*

```
import java.util.StringTokenizer;

public class EjemplosStringTokenizer {

   public static void main(String[] args) {
       String token;
       String texto1="Este es un texto de prueba";
       StringTokenizer tokenizer1=new StringTokenizer(texto1);
       while(tokenizer1.hasMoreTokens()){
          token=tokenizer1.nextToken();
          System.out.println("El token es: "+token);
       }
       String texto2="Este;es;otro;texto;de;prueba";
       StringTokenizer tokenizer2=new StringTokenizer(texto2,";");
       while(tokenizer2.hasMoreTokens()){
          token=tokenizer2.nextToken();
          System.out.println("El token con delimitador ; es: "+token);
       }
   }
}
```

Salida estándar

```
El token es: Este
El token es: es
El token es: un
El token es: texto
El token es: de
El token es: prueba
El token con delimitador ; es: Este
El token con delimitador ; es: es
El token con delimitador ; es: otro
El token con delimitador ; es: texto
El token con delimitador ; es: de
El token con delimitador ; es: prueba
```

4.7 Clase *BigInteger*

La clase *BigInteger* permite en una aplicación manejar datos numéricos enteros positivos y negativos con longitud indefinida. Considerando que el tipo primitivo de datos "*long*" tiene un valor máximo de 9.223.372.036.854.775.807, en caso de que sea necesario realizar operaciones con datos enteros mayores a este valor, se

hace estrictamente necesario el uso de la clase *BigInteger. La* clase *BigInteger* pertenece al paquete **java.math** del *API* de Java.

Tabla 20. Métodos principales de la clase *BigInteger*

Retorno	Método	Descripción
void	*BigInteger(String val)*	Constructor que inicializa el *BigInteger* con el valor de la cadena de caracteres del parámetro.
BigInteger	*abs()*	Retorna el valor absoluto del *BigInteger*.
BigInteger	*add(BigInteger val)*	Retorna la suma del *BigInteger* con el parámetro.
BigInteger	*and(BigInteger val)*	Retorna el resultado de la operación lógica *AND* del BigInteger con el parámetro.
int	*bitCount()*	Retorna el número de bits del *BigInteger*.
int	*compareTo(BigInteger val)*	Compara el *BigInteger con* el parámetro.
BigInteger	*divide(BigInteger val)*	Retorna la división del *BigInteger con* el parámetro.
BigInteger[]	*divideAndRemainder (BigInteger val)*	Retorna el cociente y residuo de la división del *BigInteger con* el parámetro.
double	*doubleValue()*	Convierte el *BigInteger* a *double.*
float	*floatValue()*	Convierte el *BigInteger a float.*
int	*intValue()*	Convierte el *BigInteger a int.*
long	*longValue()*	Convierte el *BigInteger a long.*
BigInteger	*max(BigInteger val)*	Retorna el mayor entre el *BigInteger* y el parámetro.
BigInteger	*min(BigInteger val)*	Retorna el menor entre el *BigInteger y* el parámetro.
BigInteger	*mod(BigInteger m)*	Retorna el residuo de la división del *BigInteger con* el parámetro.

BigInteger	*multiply(BigInteger val)*	Retorna la multiplicación del *BigInteger con* el parámetro.
BigInteger	*not()*	Retorna el resultado de la operación lógica *NOT* del *BigInteger.*
BigInteger	*or(BigInteger val)*	Retorna el resultado de la operación lógica *OR* del *BigInteger con* el parámetro.
BigInteger	*pow(int exponent)*	Retorna la potenciación del *BigInteger con* el parámetro.
BigInteger	*subtract (BigInteger val)*	Retorna la resta del *BigInteger con* el parámetro.
String	*toString()*	Retorna el *BigInteger* en una cadena de caracteres
static BigInteger	*valueOf(long val)*	Retorna un *BigInteger con* el valor del parámetro.
BigInteger	*xor(BigInteger val)*	Retorna el resultado de la operación lógica *XOR* del *BigInteger con* el parámetro.

Ejemplo de uso del método *pow*

```
import java.math.BigInteger;

public class EjemplosBigInteger {

   public static void main(String[] args) {
       long potLong;
       int base=10;
       for(int exp=15;exp<=25;exp++){
          potLong=(long)Math.pow(base, exp);
          System.out.println("La potenciación usando long de "+
          base+"^"+exp+" es:"+potLong);
       }
       BigInteger potBigInteger=new BigInteger(String.
valueOf(base));
       for(int exp=15;exp<=25;exp++){
          System.out.println("La potenciación usando BigInteger de "+
          base+"^"+exp+" es: "+potBigInteger.pow(exp));
       }
   }
}
```

Salida estándar
La potenciación usando long de 10^15 es:1000000000000000

```
La potenciación usando long de 10^15 es:1000000000000000
La potenciación usando long de 10^16 es:10000000000000000
La potenciación usando long de 10^17 es:100000000000000000
La potenciación usando long de 10^18 es:1000000000000000000
La potenciación usando long de 10^19 es:9223372036854775807
La potenciación usando long de 10^20 es:9223372036854775807
La potenciación usando long de 10^21 es:9223372036854775807
La potenciación usando long de 10^22 es:9223372036854775807
La potenciación usando long de 10^23 es:9223372036854775807
La potenciación usando long de 10^24 es:9223372036854775807
La potenciación usando long de 10^25 es:9223372036854775807
La potenciación usando BigInteger de 10^15 es: 1000000000000000
La potenciación usando BigInteger de 10^16 es: 10000000000000000
La potenciación usando BigInteger de 10^17 es: 100000000000000000
La potenciación usando BigInteger de 10^18 es: 1000000000000000000
La potenciación usando BigInteger de 10^19 es: 10000000000000000000
La potenciación usando BigInteger de 10^20 es: 100000000000000000000
La potenciación usando BigInteger de 10^21 es: 1000000000000000000000
La potenciación usando BigInteger de 10^22 es: 10000000000000000000000
La potenciación usando BigInteger de 10^23 es: 100000000000000000000000
La potenciación usando BigInteger de 10^24 es: 1000000000000000000000000
La potenciación usando BigInteger de 10^25 es: 10000000000000000000000000
```

4.8 Ejercicios propuestos

1. Implemente mediante la clase *String*, una aplicación que cuente el número de vocales de un texto.
2. Implemente mediante la clase *BigInteger*, una aplicación que emule una calculadora aritmética y lógica. Esta calculadora debe permitir operaciones con números muy grandes.
3. Implemente una aplicación que permita convertir números decimales a su equivalencia en cualquier base.

CAPÍTULO 5

Entrada y salida estándar

Toda aplicación requiere información de entrada para realizar las operaciones que cada proceso resuelve, así como requiere mecanismos de salida para entregar los resultados. La forma de representar estas entradas y salidas en Java es con base en "*streams*" que son flujos de datos. Un "*stream*" se comporta como un medio de comunicación entre el programa y la fuente o destino de los datos. La información se transporta en serie a través de este medio.

5.1 Clase *System*

En Java, la entrada desde teclado y la salida a pantalla se utilizan a través de la clase *System*. Esta clase pertenece al paquete **java.lang** y posee atributos y métodos que se relacionan directamente con el sistema local.

Tabla 21. Atributos de la clase *System*

Tipo	Atributo	Descripción
static InputStream	*in*	Objeto preparado para recibir datos desde la entrada estándar del sistema que generalmente es el teclado.
static PrintStream	*out*	Objeto preparado para imprimir los datos en la salida estándar del sistema que generalmente es la pantalla.
static PrintStream	*err*	Objeto utilizado para mensajes de error presentados por defecto en pantalla.

5.2 Clase *InputStream*

La clase *InputStream* es necesaria para realizar la lectura de información por entrada estándar. Esta clase pertenece al paquete **java.io**.

Tabla 22. Métodos principales de la clase *InputStream*

Retorno	Método	Descripción
int	*available()*	Retorna el número estimado de *bytes* que pueden ser leídos por el *InputStream* *Constructor* que inicializa el *BigInteger* con el valor de la cadena de caracteres del parámetro.
abstract int	*read()*	Lee el siguiente *byte* de datos del *InputStream*.
int	*read(byte[] b)*	Lee un número de *bytes* desde el *InputStream* y lo almacena en el parámetro.
int	*read(byte[] b, int off, int len)*	Lee hasta la longitud dada por el parámetro *len* desde el *InputStream* y lo almacena en el parámetro *b*.
long	*skip(long n)*	Salta y descarta *n bytes* de datos del *InputStream.*

Ejemplo de uso del método *read*

Para leer desde teclado se puede utilizar el método *read* de la clase *InputStream*. Este método lee un carácter por cada llamada. Su valor de retorno es un *int* que corresponde al código *ASCII* del

carácter capturado. Si se espera cualquier otro tipo hay que hacer una conversión explícita mediante un "*cast*". Así mismo, este método requiere realizar el manejo de excepción de la clase *IOException*.

```
import java.io.IOException;

public class EjemplosSystem {

   public static void main(String[] args) {
       char c;
       try {
          c=(char)System.in.read();
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

Para leer cadenas de datos es necesario emplear una estructura de repetición *while* o *for* y unir los caracteres.

```
import java.io.IOException;

public class EjemplosSystem {

   public static void main(String[] args) {
       char c;
       String cadena="";
       try {
          while((c=(char)System.in.read()) != '\n')
               cadena = cadena + c;
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

5.3 Clase *PrintStream*

La clase *PrintStream*, es necesaria para realizar la escritura de información por salida estándar. Esta clase pertenece al paquete **java.io.**

Tabla 23. Métodos principales de la clase *PrintStream*

Retorno	Método	Descripción
PrintStream	*append(char c)*	Agrega el carácter del parámetro al *PrintStream.*
void	*close()*	Cierra el *PrintStream*.
void	*print(String s)*	Imprime la cadena del parámetro.
void	*println()*	Imprime un salto de línea.
void	*println(String x)*	Imprime la cadena del parámetro más un salto de línea.
void	*write(int b)*	Escribe el *byte* especificado del parámetro en el *PrintStream*.

Para escribir cadenas de datos en salida estándar se requiere usar el objeto *System.out*.

```
System.out.println("Hola mundo");
```

5.4 Clase *BufferedReader*

La clase *BufferedReader*, proporciona métodos para la captura de información en cadenas de caracteres, optimizando el proceso que se debería realizar a través del objeto "*in*" de la clase *System*. Esta clase pertenece al paquete **java.io.**

Tabla 24. Métodos principales de la clase *BufferedReader*

Retorno	Método	Descripción
void	*BufferedReader (Reader in)*	Constructor que crea un buffer de entrada de datos a través del parámetro.
void	*close()*	Cierra el *Stream.*
int	*read()*	Lee un carácter y retorna el valor entero del carácter equivalente al código *ASCII.*
String	*readLine()*	Lee una cadena de caracteres.

Ejemplo de uso del método *readLine*

Una de las formas más simples de utilizar el *BufferedReader*, con entrada estándar, es a través de una estructura de repetición *while* para leer indefinidamente, hasta no encontrar más información.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EjemplosSystem {

   public static void main(String[] args) {
       String cadena="";
       BufferedReader br = new BufferedReader(new
       InputStreamReader(System.in));
       try {
          while((cadena=br.readLine())!=null){
              //Código que usa el objeto cadena
          }
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

Si se requiere capturar datos numéricos, es necesario realizar un "*cast*" y debe controlarse la correspondiente excepción.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EjemplosSystem {

   public static void main(String[] args) {
       String cadena="";
       BufferedReader br = new BufferedReader(new
       InputStreamReader(System.in));
       int dato;
       try {
          while((cadena=br.readLine())!=null){
              dato=Integer.parseInt(cadena);
              dato+=1;
              System.out.println(dato);
          }
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

Entrada estándar	Salida estándar

`10`
`20`
`30`
`40`
`50`
`Hola mundo`
	`11`
	`21`
	`31`
	`41`
	`51`
	`Exception in thread “main” java.lang.NumberFormatException: For input string: “Hola mundo”`
	`at java.lang.NumberFormatException.forInputString(Unknown Source)`
	`at java.lang.Integer.parseInt(Unknown Source)`
	`at java.lang.Integer.parseInt(Unknown Source)`
	`at EjemplosSystem.main(EjemplosSystem.java:16)`

En el caso anterior queda evidencia del uso de la excepción, debido a que el texto “Hola mundo”, no se puede convertir a numero, se produce la excepción controlada por la clase *java.lang.NumberFormatException*.

5.5 Clase *Scanner*

La clase *Scanner* proporciona métodos para la captura de información en diferentes tipos de datos. Esta clase pertenece al paquete **java.io.**

Tabla 25. Métodos principales de la clase *Scanner*

Retorno	Método	Descripción
void	*Scanner(File source)*	Constructor que crea un *Scanner* que produce valores capturados desde un archivo especificado por el parámetro.
void	*Scanner (InputStream source)*	Constructor que crea un *Scanner* que produce valores capturados desde un *InputStream* especificado por el parámetro.

void	*close()*	Cierra el *Scanner*.
boolean	*hasNext()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada.
boolean	*hasNextBigInteger()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como *BigInteger.*
boolean	*hasNextBoolean()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como booleano mediante las cadenas *true* o *false.*
boolean	*hasNextDouble()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como *double.*
boolean	*hasNextFloat()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como *float.*
boolean	*hasNextInt()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como *int.*
boolean	*hasNextLine()*	Retorna verdadero si el *Scanner* posee una nueva cadena de caracteres.
boolean	*hasNextLong()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como *long.*
boolean	*hasNextShort()*	Retorna verdadero si el *Scanner* posee otro *token* en la entrada que puede ser interpretado como *short.*
String	*next()*	Retorna el siguiente *token* como cadena de caracteres.
BigInteger	*nextBigInteger()*	Retorna el siguiente *token* como *BigInteger.*

boolean	*nextBoolean()*	Retorna el siguiente *token* como *boolean.*
double	*nextdouble()*	Retorna el siguiente *token* como *double.*
float	*nextFloat()*	Retorna el siguiente *token* como *float.*
int	*nextInt()*	Retorna el siguiente *token* como *int.*
String	*nextLine()*	Retorna la siguiente línea como cadena de caracteres.
long	*nextLong()*	Retorna el siguiente *token* como *long.*
short	*nextShort()*	Retorna el siguiente *token* como *short.*

Ejemplo de uso de los métodos *hasNext, hasNextInt, next* y *nextInt*

Conociendo los datos de entrada es de bastante utilidad la clase *Scanner*, debido a que esta no requiere realizar "*cast*" en caso en que los datos sean diferentes de *String*. Por otro lado, es sencillo controlar el tipo de dato del siguiente *token* evitando presencia de excepciones.

```
import java.util.Scanner;

public class EjemplosSystem {

   public static void main(String[] args) {
       Scanner s = new Scanner(System.in);
       int dato;
       while(s.hasNext()){
          if(s.hasNextInt()){
              dato=s.nextInt();
              dato+=1;
              System.out.println(dato);
          }else{
              System.out.println(s.next());
          }
       }
   }
}
```

Entrada estándar	Salida estándar

`10`
`20`
`30`
`40`
`50`
`Hola mundo`
	`11`
	`21`
	`31`
	`41`
	`51`
	`Hola`
	`mundo`

5.6 Ejercicios propuestos

1. Implemente una aplicación que emule una calculadora capturando información por entrada estándar.

2. Con base en la clase *Date*, implemente una aplicación que capture 1.000 datos con *BufferedReader* y calcule su tiempo en milisegundos. Implemente el mismo ejercicio con la clase *Scanner* y calcule el tiempo en milisegundos. Determine diferencias de tiempo de ejecución entre las dos clases.

CAPÍTULO 6

Arreglos, matrices y colecciones

En cualquier lenguaje de programación es posible construir estructuras que almacenen conjuntos de datos. Estas estructuras pueden tener una o más dimensiones. Las estructuras con una dimensión se denominan arreglos y las estructuras con dos dimensiones se denominan matrices. Hay casos particulares en que se requiere el uso de más de dos dimensiones, en ese caso, se denominan arreglos multidimensionales. Java además incluye en su *API* clases que proveen servicios para el almacenamiento de objetos denominadas colecciones.

6.1 Arreglos

Un arreglo es una estructura que posee un conjunto de datos del mismo tipo. Los arreglos en Java son objetos pero sus elementos pueden ser tipos primitivos de datos o clases.

Un arreglo tiene las siguientes características:

- Nombre. El nombre identifica al arreglo y a través de este, se accede al arreglo para su lectura y escritura de información.

```
int [] miArreglo;
```

- Instancia con el operador "*new*" que permite la asignación del tamaño del arreglo.

```
miArreglo = new int[20];
```

- Es posible realizar los procedimientos anteriores en una sola línea de código.

```
int [] miArreglo = new int[20];
```

- Se accede a los elementos del arreglo a través de corchetes cuadrados "[]" indicando la posición del elemento al cual se desea acceder. La posición del elemento es denominada índice. El índice inicial es *0* y el final es *n-1* donde *n* es la cantidad de elementos del arreglo.

```
miArreglo[0]=10;
```

- Los elementos de un arreglo se inicializan al valor por defecto del tipo de dato.
- Los arreglos se pueden inicializar con valores entre corchetes "{}" separados por comas.

```
String dias[] = {"lunes", "martes", "miércoles", "jueves",
"viernes", "sábado", "domingo"};
```

La Figura 3 representa un arreglo en Java:

días[0]	lunes
días[1]	martes
días[2]	miércoles
días[3]	jueves
días[4]	viernes
días[5]	sábado
días[6]	domingo

Figura 3. Representación de un arreglo en Java

6.1.1 Cálculo de promedio en un arreglo

Para calcular el promedio en un arreglo, basta con hacer la sumatoria de números del arreglo y dividir en el número de elementos.

Su implementación es la siguiente:

```
import java.util.Scanner;

public class EjemplosArreglos {

   public static void main(String[] args) {
       Scanner s = new Scanner(System.in);
       double promedio;
       double sumatoria=0;
       int tamano=10;
       double []arreglo = new double[tamano];
       for(int i=0; i<tamano; i++){
          arreglo[i]=s.nextInt();
       }
       System.out.println("El arreglo original es:");
       for(int i=0; i<tamano; i++){
          System.out.println(arreglo[i]);
       }
       for(int i=0; i<tamano; i++){
          sumatoria+=arreglo[i];
       }
       promedio=sumatoria/tamano;
       System.out.println("El promedio es:"+promedio);
   }
}
```

Entrada estándar	Salida estándar
25 34 12 8 7 11 45 20 21 10	El arreglo original es: 25.0 34.0 12.0 8.0 7.0 11.0 45.0 20.0 21.0 10.0 El promedio es:19.3

6.1.2 Búsqueda lineal

La búsqueda lineal en un arreglo consiste en la implementación de un proceso iterativo que recorre todo el arreglo. Se debe contar con un índice que inicia en *0* e incrementa en cada iteración. En cada

iteración consulta a través de un "*if*" si el valor a buscar es igual al valor del arreglo en índice actual. En caso de ser verdadero, debe retornar el índice. Si se compara todo el arreglo y en ningún caso encuentra el valor, retorna el valor *-1*.

Su implementación es la siguiente:

```
public class EjemplosArreglos {

   public static void main(String[] args) {
       int []numeros = new int[10];
       for(int i=0;i<numeros.length;i++){
          numeros[i]=i*5;
       }
       System.out.println("El arreglo original es:");
       for(int i=0;i<numeros.length;i++){
          System.out.println(numeros[i]);
       }
       EjemplosArreglos ejemplo = new EjemplosArreglos();
       int indice=ejemplo.busquedaLineal(numeros, 40);
       System.out.println("El indice del valor '40' es: "+indice);

   }

   public int busquedaLineal(int []arreglo, int clave){
       for(int i=0;i<arreglo.length;i++){
          if(arreglo[i]==clave){
              return i;
          }
       }
       return -1;
   }
}
```

Salida estándar

```
El arreglo original es:
0
5
10
15
20
25
30
35
40
45
El índice del valor '40' es: 8
```

6.1.3 Búsqueda binaria

La búsqueda binaria en un arreglo es más eficiente y sofisticada que la búsqueda lineal, y consiste en la implementación de un proceso recursivo que recibe el arreglo y el valor que se desea buscar. Esta búsqueda debe realizarse con arreglos ordenados. Se inicia consultando el valor ubicado en la mitad del arreglo. Si el valor a buscar es igual, se retorna el índice. Si el valor a buscar es menor, se hace el llamado recursivo realizando la búsqueda con los valores desde la posición inicial hasta la posición anterior a la mitad, pero si no hay elementos entre estas posiciones, retorna *-1*. Si el valor a buscar es mayor, se hace el llamado recursivo realizando la búsqueda con los valores desde la posición siguiente a la mitad hasta la última posición, pero si no hay elementos entre estas posiciones, retorna *-1*.

Este algoritmo se puede implementar sin usar recursividad, sin embargo, es recomendable implementarlo de esta forma para optimizar el código. Su implementación es la siguiente:

```
public class EjemplosArreglos {

   public static void main(String[] args) {
       int []numeros = new int[10];
       for(int i=0;i<numeros.length;i++){
          numeros[i]=i*5;
       }
       System.out.println("El arreglo original es:");
       for(int i=0;i<numeros.length;i++){
          System.out.println(numeros[i]);
       }
       EjemplosArreglos ejemplo = new EjemplosArreglos();
       int indice=ejemplo.busquedaBinaria(numeros, 40, 0,
       numeros.length-1);
       System.out.println("El indice del valor '40' es: "+
       indice);
   }

public int busquedaBinaria(int []arreglo, int clave, int
   posInicial, int posFinal){

   int posMitad=(posFinal+posInicial)/2;
   if(clave==arreglo[posMitad]){
       return posMitad;
   }else if(clave<arreglo[posMitad]){
```

```
        if(posMitad-1<=posInicial){
           return -1;
        }else{
           return busquedaBinaria(arreglo,clave,posInicial,
           posMitad-1);
        }
     }else{
        if(posMitad+1>=posFinal){
           return -1;
        }else{
           return busquedaBinaria(arreglo,clave,posMitad+1,
           posFinal);
        }
     }
  }
}
```

Salida estándar
El arreglo original es: 0 5 10 15 20 25 30 35 40 45 El índice del valor '40' es: 8

6.1.4 Ordenamiento de un arreglo de números

Considerando un arreglo tipo *int*, es posible ordenarlo utilizando diferentes técnicas que varían en su complejidad de implementación y de ejecución. La técnica más simple es el ordenamiento de burbuja.

El algoritmo de burbuja consiste en evaluar cada elemento por los demás elementos. Para ello, se requiere un índice que recorra el arreglo desde la primera posición hasta la penúltima. Adicionalmente, se requiere un índice que recorra el arreglo desde la siguiente posición del primer índice hasta la última posición del arreglo. Con base en un arreglo denominado A de 5 posiciones y los datos en cada posición

son: 8, 3, 5, 9, 1, la representación del ordenamiento mediante el algoritmo de burbuja se presenta en la Figura 4.

Iteración 1				
i=0				
8	3	5	9	1
	j=i+1=1			
A[i]>A[j]: En este caso es verdadero, entonces se realiza intercambio de los datos				
3	8	5	9	1
Iteración 2				
i=0				
3	8	5	9	1
		j++ = 2		
A[i]>A[j]: En este caso es falso				
Iteración 3				
i=0				
3	8	5	9	1
			j++ = 3	
A[i]>A[j]: En este caso es falso				
Iteración 4				
i=0				
3	8	5	9	1
				j++ = 4
A[i]>A[j]: En este caso es verdadero, entonces se realiza intercambio de los datos				
1	8	5	9	3
Iteración 5				
	i++=1			
1	8	5	9	3
		j=i+1=2		
A[i]>A[j]: En este caso es verdadero, entonces se realiza intercambio de los datos				
1	5	8	9	3

Iteración 6				
	i = 1			
1	5	8	9	3
			j++ = 3	
A[i]>A[j]: En este caso es falso				
Iteración 7				
	i = 1			
1	5	8	9	3
				j++ = 4
A[i]>A[j]: En este caso es verdadero, entonces se realiza intercambio de los datos				
1	3	8	9	5
Iteración 8				
		i++ = 2		
1	3	8	9	5
			j=i+1 = 3	
A[i]>A[j]: En este caso es falso				
Iteración 9				
		i = 2		
1	3	8	9	5
				j++ = 4
A[i]>A[j]: En este caso es verdadero, entonces se realiza intercambio de los datos				
1	3	5	9	8
Iteración 10				
			i++ = 3	
1	3	5	9	8
				j=i+1 = 4
A[i]>A[j]: En este caso es verdadero, entonces se realiza intercambio de los datos				
1	3	5	8	9

Figura 4. Representación del algoritmo de ordenamiento de burbuja

Su implementación es la siguiente:

```
import java.util.Scanner;

public class EjemplosArreglos {

   public static void main(String[] args) {
       Scanner s = new Scanner(System.in);
       int []arreglo = new int[10];
       int indice=0;
       while(indice<10){
          arreglo[indice]=s.nextInt();
          indice++;
       }
       System.out.println("El arreglo original es:");
```

```
        for(int i=0; i<indice; i++){
           System.out.println(arreglo[i]);
        }
        for(int i=0; i<indice-1; i++){
           for(int j=i+1; j<indice; j++){
              if(arreglo[i]>arreglo[j]){
                 int temporal=arreglo[i];
                 arreglo[i]=arreglo[j];
                 arreglo[j]=temporal;
              }
           }
        }
        System.out.println("El arreglo ordenado es:");
        for(int i=0; i<indice; i++){
           System.out.println(arreglo[i]);
        }
    }
}
```

Entrada estándar	Salida estándar
25 34 12 8 7 9 45 20 21 10	El arreglo original es: 25 34 12 8 7 9 45 20 21 10 El arreglo ordenado es: 7 8 9 10 12 20 21 25 34 45

6.2 Matrices

Una matriz es un arreglo bidimensional. Una matriz cuenta con filas y columnas. Con base en la combinación de fila y columna, se puede acceder a cada uno de los elementos de la matriz.

Una matriz tiene las siguientes características:

- Nombre. El nombre identifica al arreglo y a través de este, se accede al arreglo para su lectura y escritura de información.

```
int [][] miMatriz;
```

- Instancia con el operador "*new*" que permite la asignación del tamaño del arreglo.

```
miMatriz = new int[3][4];
```

- Es posible realizar los procedimientos anteriores en una sola línea de código.

```
int [][] miMatriz = new int[3][4];
```

- Los elementos de una matriz se inicializan al valor por defecto del tipo de dato.
- Las matrices se pueden inicializar con valores entre corchetes "{}" separados por comas por cada fila de datos. Cada fila también se separa por comas.

```
int [][] miMatriz ={{1,2,3,4},
   {5,6,7,8},
   {9,10,11,12}
}
```

La Figura 5 representa una matriz en Java:

	Columna 0	Columna 1	Columna 2	Columna 3
Fila 0	elemento[0][0]	elemento[0][1]	elemento[0][2]	elemento[0][3]
Fila 1	elemento[1][0]	elemento[1][1]	elemento[1][2]	elemento[1][3]
Fila 2	elemento[2][0]	elemento[2][1]	elemento[2][2]	elemento[2][3]

Figura 5. Representación de una matriz en Java

6.2.1 Cálculo de la traspuesta de una matriz

La traspuesta de una matriz consiste en intercambiar filas por columnas y viceversa. Su implementación es la siguiente:

```
public class EjemplosMatrices {

   public static void main(String[] args) {
       int [][] miMatriz ={{1,2,3},
          {4,5,6},
          {7,8,9}
       };
       int [][] miMatrizTraspuesta = new int[3][3];
       System.out.println("La matriz original es:");
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              System.out.print(miMatriz[i][j]+"\t");
          }
          System.out.println();
       }
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              miMatrizTraspuesta[j][i]=miMatriz[i][j];
          }
       }
       System.out.println("La matriz traspuesta es:");
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              System.out.print(miMatrizTraspuesta[i][j]+"\t");
          }
          System.out.println();
       }
   }
}
```

Salida estándar

```
La matriz original es:
1       2       3
4       5       6
7       8       9
La matriz traspuesta es:
1       4       7
2       5       8
3       6       9
```

6.2.2 Multiplicación de matrices

Dadas dos matrices A y B donde el número de columnas de la matriz A es igual al número de filas de la matriz B que se denota como:

$A: = (a_{ij})_{m \times n}$ y $B: = (b_{ij})_{n \times p}$

La multiplicación de *A x B* se denota como:

$AxB: = (c_{ij})_{m \times p}$

Donde cada elemento c_{ij} se calcula mediante:

$$c_{ij} = \sum_{z=0}^{n-1} a_{iz} \times b_{zj}$$

Para dos matrices 3 x 3, la representación gráfica de la multiplicación se presenta en la Figura 6:

A[0][0]	A[0][1]	A[0][2]		B[0][0]	B[0][1]	B[0][2]	
A[1][0]	A[1][1]	A[1][2]	x	B[1][0]	B[1][1]	B[1][2]	=
A[2][0]	A[2][1]	A[2][2]		B[2][0]	B[2][1]	B[2][2]	

A[0][0]*B[0][0]+ A[0][1]*B[1][0]+ A[0][2]*B[2][0]	A[0][0]*B[0][1]+ A[0][1]*B[1][1]+ A[0][2]*B[2][1]	A[0][0]*B[0][2]+ A[0][1]*B[1][2]+ A[0][2]*B[2][2]
A[1][0]*B[0][0]+ A[1][1]*B[1][0]+ A[1][2]*B[2][0]	A[1][0]*B[0][1]+ A[1][1]*B[1][1]+ A[1][2]*B[2][1]	A[1][0]*B[0][2]+ A[1][1]*B[1][2]+ A[1][2]*B[2][2]
A[2][0]*B[0][0]+ A[2][1]*B[1][0]+ A[2][2]*B[2][0]	A[2][0]*B[0][1]+ A[2][1]*B[1][1]+ A[2][2]*B[2][1]	A[2][0]*B[0][2]+ A[2][1]*B[1][2]+ A[2][2]*B[2][2]

Figura 6. Representación de multiplicación de dos matrices

La implementación para dos matrices de 3 x 3 es la siguiente:

```
public class EjemplosMatrices {

   public static void main(String[] args) {
       int [][] A ={{1,2,3},
          {4,5,6},
          {7,8,9}
       };
       int [][] B ={{9,8,7},
          {6,5,4},
          {3,2,1}
       };
       int [][] C = new int[3][3];
       System.out.println("La matriz A es:");
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              System.out.print(A[i][j]+"\t");
          }
          System.out.println();
       }
       System.out.println("La matriz B es:");
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              System.out.print(B[i][j]+"\t");
          }
          System.out.println();
       }
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              for(int k=0; k<3; k++){
                 C[i][j]+=A[i][k]*B[k][j];
              }
          }
       }
       System.out.println("La multiplicación es:");
       for(int i=0; i<3; i++){
          for(int j=0; j<3; j++){
              System.out.print(C[i][j]+"\t");
          }
          System.out.println();
       }
   }
}
```

Salida estándar

```
La matriz A es:
1       2       3
4       5       6
7       8       9
La matriz B es:
9       8       7
6       5       4
3       2       1
La multiplicación es:
30      24      18
84      69      54
138     114     90
```

6.3 Clase *Vector*

La clase *Vector* es una de las clases de Java que sirve para trabajar colecciones de objetos. Esta clase pertenece al paquete **java.util** y deriva de *Object*. La clase *Vector* permite representar un arreglo de objetos tipo *Object*. Así mismo permite acceder a los elementos con un índice.

Tabla 26. Métodos principales de la clase *Vector*

Retorno	Método	Descripción
void	*Vector()*	Constructor que crea un *Vector* vacío con un tamaño por defecto de 10 elementos.
void	*Vector(int initial Capacity)*	Constructor que crea un *Vector* vacío con un tamaño definido por el parámetro.
boolean	*add(Object e)*	Agrega el objeto del parámetro al final del *Vector*.
void	*add(int index, Object element)*	Inserta el objeto del segundo parámetro en la posición dada por el primer parámetro.

void	*addElement (Object obj)*	Adiciona el objeto del parámetro al final de la lista e incrementa el tamaño del *Vector* en uno.
int	*capacity()*	Retorna la capacidad actual del *Vector*.
void	*clear()*	Retira todos los elementos del *Vector*.
boolean	*contains(Object o)*	Retorna verdadero si el *Vector* contiene el objeto del parámetro.
Object	*elementAt(int index)*	Retorna el objeto de la posición especificada en el parámetro.
Object	*firstElement()*	Retorna el objeto de la primera posición del *Vector*.
Object	*get(int index)*	Retorna el objeto de la posición especificada en el parámetro.
int	*indexOf(Object o)*	Retorna el índice de la primera ocurrencia del objeto del parámetro o retorna *-1* si el *Vector* no contiene dicho objeto.
void	*insertElementAt (Object obj, int index)*	Inserta el objeto del primer parámetro en el índice especificado en el segundo parámetro.
boolean	*isEmpty()*	Retorna verdadero si el *Vector* no contiene elementos.
Object	*lastElement ()*	Retorna el objeto de la última posición del *Vector*.
int	*lastIndexOf (Object o)*	Retorna el índice de la última ocurrencia del objeto del parámetro o retorna *-1* si el *Vector* no contiene dicho objeto.

boolean	*remove(Object o)*	Remueve la primera ocurrencia del objeto especificado en el *Vector*. Si el *Vector* no contiene el elemento retorna falso.
void	*removeAll Elements()*	Remueve todos los objetos del *Vector* y coloca su tamaño en cero.
void	*removeElementAt (int index)*	Elimina el objeto de la posición especificada en el parámetro.
void	*setElementAt (Object obj, int index)*	Coloca el objeto del primer parámetro en la posición especificada en el segundo parámetro.
void	*setSize(int newSize)*	Coloca tamaño al *Vector*.
int	*size()*	Retorna el número de objetos que hay en el *Vector*.

Ejemplo de uso de los métodos *size, add, elementAt, insertElementAt* y *remove*

Los elementos que se almacenan en un *Vector* son objetos instancia de la clase *Object*. Sin embargo, todas las clases en Java se derivan de dicha clase. Esto indica que no es posible almacenar en un *Vector*, un tipo primitivo de dato. Al recuperar un objeto de un *Vector*, es necesario realizar el "*casting*" correspondiente para hacer uso del objeto.

```
import java.util.Vector;

public class EjemplosVector {

   public static void main(String[] args) {
       Vector v = new Vector();
       for(int i=1;i<=10;i++){
          v.add(new String("Mensaje "+i));
       }
```

```
        System.out.println("Elementos del vector:");
        String mensaje;
        for(int i=0;i<v.size();i++){
           mensaje=(String)v.elementAt(i);
           System.out.println(mensaje);
        }

        v.insertElementAt(new String("Mensaje insertado"), 5);
        System.out.println("Elementos del vector después de
        insertar:");
        for(int i=0;i<v.size();i++){
           mensaje=(String)v.elementAt(i);
           System.out.println(mensaje);
        }

        v.remove(0);
        System.out.println("Elementos del vector después de
        remover:");
        for(int i=0;i<v.size();i++){
           mensaje=(String)v.elementAt(i);
           System.out.println(mensaje);
        }
    }
}
```

Salida estándar

```
Elementos del vector:
Mensaje 1
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
Elementos del vector después de insertar:
Mensaje 1
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje insertado
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
```

```
Elementos del vector después de remover:
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje insertado
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
```

6.4 Clase *ArrayList*

La clase *ArrayList* es similar a la clase *Vector*. Sirve para trabajar colecciones de objetos. La clase *ArrayList* pertenece al paquete **java. util** y deriva de *Object*. La clase *ArrayList* permite representar un arreglo de objetos de un Tipo Abstracto de Dato. Así mismo permite acceder a los elementos con un índice.

Tabla 27. Métodos principales de la clase *ArrayList*

Retorno	Método	Descripción
void	*ArrayList()*	Constructor que crea un *ArrayList* vacío con un tamaño por defecto de 10 elementos.
void	*ArrayList (int initialCapacity)*	Constructor que crea un *ArrayList* vacío con un tamaño definido por el parámetro.
boolean	*add(E element)*	Agrega el objeto del parámetro al final del *ArrayList*. El objeto es de tipo E.
void	*add(int index, E element)*	Inserta el objeto del segundo parámetro en la posición dada por el primer parámetro.
int	*capacity()*	Retorna la capacidad actual del *ArrayList*.

void	*clear()*	Retira todos los elementos del *ArrayList*.
boolean	*contains(Object o)*	Retorna verdadero si el *ArrayList* contiene el objeto del parámetro.
E	*get(int index)*	Retorna el objeto de la posición especificada en el parámetro.
int	*indexOf(Object o)*	Retorna el índice de la primera ocurrencia del objeto del parámetro o retorna *-1* si el *ArrayList* no contiene dicho objeto.
boolean	*isEmpty()*	Retorna verdadero si el *ArrayList* no contiene elementos.
int	*lastIndexOf(Object o)*	Retorna el índice de la última ocurrencia del objeto del parámetro o retorna *-1* si el *ArrayList* no contiene dicho objeto.
boolean	*remove(Object o)*	Remueve la primera ocurrencia del objeto especificado en el *ArrayList*. Si el *ArrayList* no contiene el elemento retorna falso.
E	*remove(int index)*	Elimina y retorna el objeto de la posición especificada en el parámetro.
int	*size()*	Retorna el número de objetos que hay en el *ArrayList*.

Ejemplo de uso de los métodos *size, add, get* y *remove*

En este ejemplo, los elementos que se almacenan en un *ArrayList* son objetos instancia de la clase *String,* gracias a que se ha definido esta clase entre signos <>. Al recuperar un objeto de un *ArrayList*, no es necesario realizar el "*casting*" debido a que los objetos se almacenan como instancia de la clase definida y no como instancia de *Object*.

```
import java.util.ArrayList;

public class EjemplosArrayList {

   public static void main(String[] args) {
       ArrayList<String> array = new ArrayList<String>();
       for(int i=1;i<=10;i++){
          array.add("Mensaje "+i);
       }
       System.out.println("Elementos del array:");
       String mensaje;
       for(int i=0;i<array.size();i++){
          mensaje=array.get(i);
          System.out.println(mensaje);
       }

       array.add(5, "Mensaje insertado");
       System.out.println("Elementos del arraydespués de
       insertar:");
       for(int i=0;i<array.size();i++){
          mensaje=array.get(i);
          System.out.println(mensaje);
       }

       array.remove(0);
       System.out.println("Elementos del arraydespués de
       remover:");
       for(int i=0;i<array.size();i++){
          mensaje=array.get(i);
          System.out.println(mensaje);
       }
    }
}
```

Salida estándar

```
Elementos del array:
Mensaje 1
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
Elementos del array después de insertar:
Mensaje 1
Mensaje 2
```

```
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje insertado
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
Elementos del array después de remover:
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje insertado
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
```

6.5 Clase *Arrays*

La clase *Arrays* es una clase de utilidad introducida en el JDK 1.2 que contiene métodos *static* para manipular arreglos. Permite también ver los arreglos como listas. Esta clase proporciona métodos de alto nivel para realizar búsquedas, ordenamiento y llenado, entre otras funciones; para arreglos con tipos primitivos de datos y arreglos de objetos. La clase *Arrays* pertenece al paquete **java.util**.

Tabla 28. Métodos principales de la clase *Arrays*

Retorno	Método	Descripción
static int	*binarySearch(double[] a, double key)*	Busca en el arreglo del primer parámetro, el elemento *double* del segundo parámetro mediante búsqueda binaria y retorna su índice

static int	*binarySearch(double[] a, int fromIndex, int toIndex, double key)*	Busca en el arreglo del primer parámetro, en el rango especificado en el segundo y tercer parámetro, el elemento *double* del cuarto parámetro mediante búsqueda binaria y retorna su índice.
static int	*binarySearch(int[] a, int key)*	Busca en el arreglo del primer parámetro, el elemento *int* del segundo parámetro mediante búsqueda binaria y retorna su índice.
static int	*binarySearch(int[] a, int fromIndex, int toIndex, int key)*	Busca en el arreglo del primer parámetro, en el rango especificado en el segundo y tercer parámetro, el elemento *int* del cuarto parámetro mediante búsqueda binaria y retorna su índice.
static int	*binarySearch(Object[] a, Object key)*	Busca en el arreglo del primer parámetro, el elemento *Object* del segundo parámetro mediante búsqueda binaria y retorna su índice.
static int	*binarySearch(Object [] a, int fromIndex, int toIndex, Object key)*	Busca en el arreglo del primer parámetro, en el rango especificado en el segundo y tercer parámetro, el elemento *Object* del cuarto parámetro mediante búsqueda binaria y retorna su índice.

static double[]	*copyOf(double[] original, int newLength)*	Copia el arreglo *double* en la longitud especificada.
static int[]	*copyOf(int[] original, int newLength)*	Copia el arreglo *int* en la longitud especificada.
static Object[]	*copyOf(Object[] original, int newLength)*	Copia el arreglo *Object* en la longitud especificada.
static boolean	*equals(double[] a, double[] a2)*	Retorna verdadero si los arreglos *double* son iguales.
static boolean	*equals(int[] a, int[] a2)*	Retorna verdadero si los arreglos *int* son iguales.
static boolean	*equals(Object[] a, Object[] a2)*	Retorna verdadero si los arreglos *Object* son iguales.
static void	*fill(double[] a, double val)*	Llena el arreglo *double* con el valor del segundo parámetro.
static void	*fill(double[] a, int fromIndex, int toIndex, double val)*	Llena el arreglo *double* en el rango del segundo y tercer parámetro con el valor del cuarto parámetro.
static void	*fill(int[] a, int val)*	Llena el arreglo *int* con el valor del segundo parámetro.
static void	*fill(int[] a, int fromIndex, int toIndex, int val)*	Llena el arreglo *int* en el rango del segundo y tercer parámetro con el valor del cuarto parámetro.
static void	*fill(Object[] a, Object val)*	Llena el arreglo *Object* con el valor del segundo parámetro.
static void	*fill(Object[] a, int fromIndex, int toIndex, Object val)*	Llena el arreglo *Object* en el rango del segundo y tercer parámetro con el valor del cuarto parámetro.

static void	*sort(double[] a)*	Ordena el arreglo double en orden ascendente numéricamente
static void	*sort(int[] a)*	Ordena el arreglo *int* en orden ascendente numéricamente.
static void	*sort(Object[] a)*	Ordena el arreglo *Object* en orden ascendente de acuerdo a su ordenamiento natural de elementos.

Ejemplo de uso de los métodos *sort* y *copyOf*

```
import java.util.Arrays;

public class EjemplosArrays {

   public static void main(String[] args) {
       double []numeros = new double[10];
       for(int i=0;i<10;i++){
          numeros[i]=Math.round(Math.random()*1000);
       }
       System.out.println("El arreglo original es: ");
       for(int i=0;i<numeros.length;i++){
          System.out.println(numeros[i]);
       }
          Arrays.sort(numeros);
          System.out.println("El arreglo ordenado es: ");
          for(int i=0;i<numeros.length;i++){
              System.out.println(numeros[i]);
       }
       double [] numeros2=Arrays.copyOf(numeros, 5);
       System.out.println("El arreglo copiado con 5
       posiciones es: ");
       for(int i=0;i<numeros2.length;i++){
          System.out.println(numeros2[i]);
       }
   }
}
```

Salida estándar
El arreglo original es: 423.0 725.0 723.0 478.0 89.0 610.0 897.0 791.0 748.0 882.0 El arreglo ordenado es: 89.0 423.0 478.0 610.0 723.0 725.0 748.0 791.0 882.0 897.0 El arreglo copiado con 5 posiciones es: 89.0 423.0 478.0 610.0 723.0

6.6 Clase *HashTable*

La clase *HashTable* implementa una tabla que relaciona una clave con un valor. Tanto la clave como el valor pueden ser instancia de cualquier objeto diferente de *null*. La clase *HashTable* pertenece al paquete **java.util.**

Tabla 29. Métodos principales de la clase *HashTable*

Retorno	Método	Descripción
void	*clear()*	Borra el *HashTable.*

boolean	*containsKey(Object key)*	Retorna verdadero si existe la clave del parámetro en el *HashTable.*
boolean	*containsValue(Object value)*	Retorna verdadero si existe el valor del parámetro en el *HashTable*.
Object	*get(Object key)*	Retorna el valor que está relacionado con la clave del parámetro.
boolean	*isEmpty()*	Retorna verdadero si el *HashTable* está vacío.
Object	*put(K key, V value)*	Coloca en el *HashTable* un registro con la clave y valor de los parámetros.
protected void	*rehash()*	Incrementa la capacidad del *HashTable* y reorganiza los objetos para realizar el acceso más eficiente.
Object	*remove(Object key)*	Remueve la clave y valor relacionados con la clave del parámetro. Retorna el valor.
int	*size()*	Retorna el número de claves en el *HashTable*.
Collection <Object>	*values()*	Retorna una colección de los valores contenidos en el *HashTable*.

Ejemplo de uso de los métodos *put* y *get*

```
import java.util.Hashtable;

public class EjemplosHashTable {

   public static void main(String[] args) {
       Hashtable tabla = new Hashtable();
       tabla.put("Hola", "Hello");
       tabla.put("Mundo", "World");
       String clave;
       clave="Hola";
       System.out.println("La palabra "+clave+" en inglés es:"+
```

```
        tabla.get(clave));
        clave=”Mundo”;
        System.out.println(“La palabra “+clave+” en inglés es: “+
        tabla.get(clave));
    }
}
```

Salida estándar
La palabra Hola en inglés es: Hello La palabra Mundo en inglés es: World

6.7 Interfaz *Iterator*

La interfaz *Iterator,* permite realizar recorridos sobre cualquier tipo de colecciones. Esta interfaz permite hacer implementaciones estándar sobre recorridos, permitiendo modificaciones de los objetos almacenados de las instancias de las diferentes clases que implementan colecciones. La interfaz *Iterator* pertenece al paquete **java.util.**

Tabla 30. Métodos principales de la interfaz *Iterator*

Retorno	Método	Descripción
boolean	*hasNext()*	Retorna verdadero si el *Iterator* tiene más elementos almacenados.
Object	*next()*	Retorna el siguiente objeto en la iteración.
void	*remove()*	Remueve de la colección que se recorre a través del *Iterator* el último objeto retornando con el método *next*. Este método es la forma más segura de remover un elemento de una colección mientras está siendo recorrida.

Ejemplo de uso de los métodos *hasNext, next* y *remove* usando *Vector*

```
import java.util.Iterator;
import java.util.Vector;
```

```
public class EjemplosIterator {

   public static void main(String[] args) {
       Vector v = new Vector();
       for(int i=1;i<=10;i++){
          v.add(new String("Mensaje "+i));
       }
       System.out.println("Elementos del vector recorridos por
       Iterator:");
       String mensaje;
       Iterator iterator = v.iterator();
       while(iterator.hasNext()){
          mensaje=(String)iterator.next();
          System.out.println(mensaje);
       }
       iterator.remove();
       System.out.println("Elementos después de remover
       Iterator:");
       iterator = v.iterator();
       while(iterator.hasNext()){
          mensaje=(String)iterator.next();
          System.out.println(mensaje);
       }
   }
}
```

Salida estándar

```
Elementos del vector recorridos por Iterator:
Mensaje 1
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
Elementos después de remover Iterator:
Mensaje 1
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
```

Ejemplo de uso de los métodos *hasNext, next* y *remove* usando *ArrayList*

En caso de usar la interfaz *iterator* con *ArrayList*, se puede asignar el tipo abstracto de dato que contiene el *iterator* entre los signos <>, evitando realizar el *casting*.

```
import java.util.ArrayList;
import java.util.Iterator;

public class EjemplosIterator {
   public static void main(String[] args) {
       ArrayList<String> array = new ArrayList<String>();
       for(int i=1;i<=10;i++){
          array.add("Mensaje "+i);
       }
       System.out.println("Elementos del array recorridos por
       Iterator:");
       Iterator<String> iterator = array.iterator();
       while(iterator.hasNext()){
          System.out.println(iterator.next());
       }
       iterator.remove();
       System.out.println("Elementos después de remover
       Iterator:");
       iterator = array.iterator();
       while(iterator.hasNext()){
          System.out.println(iterator.next());
       }
   }
}
```

Salida estándar

```
Elementos del array recorridos por Iterator:
Mensaje 1
Mensaje 2
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
Mensaje 10
Elementos después de remover Iterator:
Mensaje 1
Mensaje 2
```

```
Mensaje 3
Mensaje 4
Mensaje 5
Mensaje 6
Mensaje 7
Mensaje 8
Mensaje 9
```

6.8 Iteración de colecciones mediante ciclo *for*

La funcionalidad de iteración también puede ser lograda usando simplemente *ciclo for*. De esta forma es necesario que la colección tenga definida el tipo abstracto de dato entre signos <>. La sintaxis de este *ciclo for* es:

```
for(ClasenombreObjeto : nombreColeccion){
   ..
}
```

Ejemplo de iteración con *ciclo for*

```
import java.util.ArrayList;

public class EjemplosIterator {
   public static void main(String[] args) {
       ArrayList<String> array = new ArrayList<String>();
       for(int i=1;i<=10;i++){
          array.add("Mensaje "+i);
       }
       System.out.println("Elementos del array recorridos por
       Iterator:");
       for(String s : array){
          System.out.println(s);
       }
   }
```

Salida estándar

`Elementos del array recorridos por Iterator:`
`Mensaje 1`
`Mensaje 2`
`Mensaje 3`
`Mensaje 4`
`Mensaje 5`
`Mensaje 6`
`Mensaje 7`
`Mensaje 8`
`Mensaje 9`
`Mensaje 10`

6.9 Ejercicios propuestos

1. Implemente una aplicación que realice la multiplicación de un arreglo con una matriz.

2. Implemente una aplicación que realice la multiplicación de dos matrices en donde el usuario escoge el número de filas y de columnas de cada matriz. Para llevar a cabo la multiplicación, el número de columnas de la primer matriz debe ser igual al número de filas de la segunda. En caso de no cumplirse esta restricción, la aplicación debe lanzar una excepción.

3. Realice una aplicación que emule la tabla de posiciones de un campeonato de fútbol. Utilice arreglos para incluir el nombre de los equipos y matrices para incluir el resultado de cada equipo considerando partidos jugados, partidos ganados, partidos empatados, partidos perdidos, goles a favor, goles en contra, promedio de goles y puntos.

CAPÍTULO 7

Escritura y lectura de archivos

La escritura y lectura de archivos en Java se pueden realizar con un manejo muy similar al de salidas y entradas estándar.

Java provee las siguientes clases:

- Para representar un archivo
  - File
- Para lectura y escritura en archivos de texto
  - FileReader
  - FileWriter
- Para lectura y escritura en bytes
  - FileInputStream
  - FileOutputStream

Se puede construir un objeto de cualquiera de estas clases a partir de un String que contenga el nombre o la dirección en disco del archivo o con un objeto instancia de la clase File que representa dicho archivo.

7.1 Clase *File*

La clase *File* permite representar un archivo o directorio. Esta clase provee información acerca del archivo o directorio en disco. La clase

File no abre archivos ni proporciona servicios para procesar archivos. Al crear un objeto de la clase *File* es necesario indicar el nombre del archivo o su ruta. Si se proporciona el nombre, Java intentará encontrar el archivo en el directorio donde se ejecuta la aplicación de Java. La clase *File* pertenece al paquete **java.io.**

Tabla 31. Métodos principales de la clase *File*

Retorno	Método	Descripción
void	*File(String pathname)*	Constructor que crea un *File* abriendo el archivo especificado en el parámetro.
boolean	*canExecute()*	Retorna verdadero si la aplicación puede ejecutar el archivo denotado en la ruta del *File.*
boolean	*canRead ()*	Retorna verdadero si la aplicación puede leer el archivo denotado en la ruta del *File.*
boolean	*canWrite ()*	Retorna verdadero si la aplicación puede escribir en el archivo denotado en la ruta del *File.*
int	*compareTo(File pathname)*	Compara la ruta del File con la del parámetro alfabéticamente. Si son iguales retorna 0.
boolean	*createNewFile()*	Crea un archivo vacío si el archivo con el nombre del *File* no existe.
boolean	*delete()*	Elimina el archivo denotado en la ruta del *File.*
boolean	*exists()*	Retorna verdadero si el archivo denotado en la ruta del *File* existe.
File	*getAbsoluteFile()*	Retorna la forma absoluta de la ruta del archivo.
File	*getCanonicalFile ()*	Retorna la forma canónica de la ruta del archivo.

String	*getName()*	Retorna el nombre del archivo o directorio representado en el *File.*
boolean	*isDirectory()*	Retorna verdadero si la ruta que representa el *File* es un directorio.
boolean	*isFile()*	Retorna verdadero si la ruta que representa el *File* es un archivo.
boolean	*isHidden()*	Retorna verdadero si el archivo denotado en la ruta del *File* está oculto.
long	*lastModified()*	Retorna el tiempo en el archivo denotado en la ruta del *File* fue modificado.
long	*length()*	Retorna el tamaño en bytes del archivo denotado en la ruta del *File.*
boolean	*mkdir()*	Crea un directorio llamado con el nombre y ruta denotado en el *File.*
boolean	*setReadOnly()*	Establece como archivo de solo lectura el archivo denotado en la ruta del *File.*
boolean	*setWritable(boolean writable)*	Establece como archivo escribible el archivo denotado en la ruta del *File.*

Ejemplo de uso de los métodos *createNewFile, getAbsoluteFile, getName, isDirectory, isFile, lastModified* y *length*

```
import java.io.File;
import java.io.IOException;
import java.util.Date;

public class EjemplosArchivos {

   public static void main(String[] args) {
```

```
        File archivo = new File("archivo.txt");
        System.out.println("El nombre del archivo es:"+
        archivo.getName());
        System.out.println("La ruta absoluta del archivo es: "+
        archivo.getAbsoluteFile().toString());
        try {
           if(archivo.createNewFile()){
               System.out.println("El archivo fue creado");;
           }else{
               System.out.println("El archivo NO fue creado");;
           }
        } catch (IOException e) {
           e.printStackTrace();
        }
        System.out.println("Es directorio? "+
        archivo.isDirectory());
        System.out.println("Es archivo? "+archivo.isFile());
        System.out.println("Fecha de modificación: "+
        archivo.lastModified());
        Date fecha = new Date(archivo.lastModified());
        System.out.println("Fecha de modificación: "+
        fecha.toString());
        System.out.println("Tamaño "+archivo.length());
    }
}
```

Salida estándar

```
El nombre del archivo es:archivo.txt
La ruta absoluta del archivo es: D:\eclipse\workspace\archivo.
txt
El archivo fue creado
Es directorio? false
Es archivo? true
Fecha de modificación: 1278975592406
Fecha de modificación: Mon Jul 12 22:59:52 GMT 2010
Tamaño 0
```

7.2 Archivos secuenciales

Los archivos secuenciales o también llamados archivos de texto plano, permiten la lectura de su información en cualquier editor de texto.

7.2.1 Escritura en archivo secuencial

La escritura en un archivo secuencial puede hacerse mediante el uso de las clases *File*, *FileWriter* y *BufferedWriter.*

La clase *BufferedWriter* requiere un objeto instancia de la clase *FileWriter* para poder abrir el archivo. A su vez, opcionalmente, la clase *FileWriter* puede requerir un objeto instancia de la clase *File*. Si no se desea usar un objeto de la clase *File*, al crear el objeto de la clase *FileWriter*, se debe incluir la ruta del archivo.

Es importante tener en cuenta que si se intenta abrir un archivo con la clase *FileWriter* y este no existe, entonces Java lo crea.

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;

public class EjemplosArchivos {

   public static void main(String[] args) {
       try {
          File archivo = new File("archivo.txt");
          FileWriter fw = new FileWriter(archivo);
          BufferedWriter bw = new BufferedWriter(fw);
          for(int i=1; i<=10; i++){
              bw.write("Escritura en archivo # "+i);
              bw.newLine();
          }
          bw.close();
       } catch (FileNotFoundException e) {
          e.printStackTrace();
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

Archivo "archivo.txt"
Escritura en archivo # 1 Escritura en archivo # 2 Escritura en archivo # 3 Escritura en archivo # 4 Escritura en archivo # 5 Escritura en archivo # 6 Escritura en archivo # 7 Escritura en archivo # 8 Escritura en archivo # 9 Escritura en archivo # 10

7.2.2 Lectura en archivo secuencial

Para realizar la lectura en un archivo secuencial se puede hacer mediante el uso de las clases *File*, *FileReader* y *BufferedReader*.

La clase *BufferedReader* requiere un objeto instancia de la clase *FileReader* para poder abrir el archivo. A su vez opcionalmente, la clase *FileReader* puede requerir un objeto instancia de la clase *File*. Si no se desea usar un objeto de la clase *File*, al crear el objeto de la clase *FileReader*, se debe incluir la ruta del archivo.

Es importante tener en cuenta que si se intenta abrir un archivo con la clase *FileReader* y este no existe, entonces se presenta una excepción controlada por la clase *FileNotFoundException*.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class EjemplosArchivos {

   public static void main(String[] args) {
       try {
          File archivo = new File("archivo.txt");
          FileReader fr = new FileReader(archivo);
          BufferedReader br = new BufferedReader(fr);
          String cadena;
          while((cadena=br.readLine())!=null){
```

```
                System.out.println(cadena);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Salida estándar

```
Texto en archivo # 1
Texto en archivo # 2
Texto en archivo # 3
Texto en archivo # 4
Texto en archivo # 5
Texto en archivo # 6
Texto en archivo # 7
Texto en archivo # 8
Texto en archivo # 9
Texto en archivo # 10
```

7.3 Archivos serializables

Los archivos serializables tienen la característica de poder almacenar objetos y no texto. Un archivo serializable lo crea una aplicación y esta misma, debe estar en la capacidad de leer e interpretar dicho archivo.

Los objetos en un archivo se visualizan como un conjunto de caracteres que no son legibles, es decir, no forman ningún mensaje consistente. Sin embargo, para la aplicación que crea el archivo, estos caracteres son directamente el objeto.

Para poder almacenar un objeto en un archivo serializable, la clase debe implementar la interfaz *Serializable* que no define ningún método. Casi todas las clases del *API* de Java son serializables.

Considerando la siguiente clase *Persona* como serializable, los objetos que sean instancia de ella pueden ser almacenados en un archivo serializable.

```
import java.io.Serializable;

public class Persona implements Serializable {
   private int id;
   private String nombre;
   private String apellido;
   private String correo;

   public Persona(int id, String nombre, String apellido,
   String correo) {
       this.id = id;
       this.nombre = nombre;
       this.apellido = apellido;
       this.correo = correo;
   }

   public int getId() {
       return id;
   }

   public void setId(int id) {
       this.id = id;
   }

   public String getNombre() {
       return nombre;
   }

   public void setNombre(String nombre) {
       this.nombre = nombre;
   }

   public String getApellido() {
       return apellido;
   }

   public void setApellido(String apellido) {
       this.apellido = apellido;
   }

   public String getCorreo() {
       return correo;
   }

   public void setCorreo(String correo) {
       this.correo = correo;
   }

   public String toString(){
       return
```

```
        "id:"+this.id+
        "; Nombre: "+this.nombre+
        "; Apellido: "+this.apellido+
        "; Correo: "+this.correo;
    }
}
```

7.3.1 Escritura en archivo serializable

Para realizar la lectura en un archivo serializable, se puede hacer mediante el uso de las clases *File*, *FileOutputStream* y *ObjectOutputStream*.

La clase *ObjectOutputStream* requiere un objeto instancia de la clase *FileOutputStream* para poder abrir el archivo. A su vez opcionalmente, la clase *FileOutputStream* puede requerir un objeto *instancia* de la clase *File*. Si no se desea usar un objeto de la clase *File*, al crear el objeto de la clase *FileOutputStream*, se debe incluir la ruta del archivo.

Es importante tener en cuenta que si se intenta abrir un archivo con la clase *ObjectOutputStream* y este no existe, entonces Java lo crea.

Con base en la clase *Persona* que implementa serializable, se hace la siguiente implementación.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class EjemplosArchivos {

   public static void main(String[] args) {
       Persona p;
       try {
          File archivo = new File("archivo.txt");
          FileOutputStream fos = new FileOutputStream(archivo);
          ObjectOutputStream oos = new ObjectOutputStream(fos);
          p=new Persona(1,"Hector","Florez",
          "hf@hectorflorez.com");
          oos.writeObject(p);
          p=new Persona(2,"Arturo","Fernandez",
```

```
            "af@hectorflorez.com");
            oos.writeObject(p);
            p=new Persona(3,"Juan","Valdez","jv@hectorflorez.com");
            oos.writeObject(p);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Archivo "archivo.txt"
¬í sr Persona£3kùV÷ I idL apellidot Ljava/lang/String;L correoq ~ L nombreq ~ xp t Florezt hf@hectorflorez.comt Hectorsq ~ t Fernandezt af@ hectorflorez.comt Arturosq ~ t Valdezt jv@ hectorflorez.comt Juan

El contenido del archivo "*archivo.txt*" es como se muestra en el cuadro anterior. Se puede apreciar que esta información no se puede interpretar abriendo dicho archivo.

En el caso anterior se ha replicado la línea de código que usa el método *writeObject*. Una solución interesante es utilizar la clase *Vector,* la cual también implementa *Serializable*. De esta forma, al escribir en el archivo, se escribe un *Vector* el cual contiene muchos objetos.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Vector;

public class EjemplosArchivos {

    public static void main(String[] args) {
        Persona p;
        Vector v = new Vector();
        p=new Persona(1,"Héctor","Flórez","hf@hectorflorez.com");
        v.add(p);
        p=new Persona(2,"Arturo","Fernández",
```

```
            "af@hectorflorez.com");            v.add(p);
            p=new Persona(3,"Juan","Valdez","jv@hectorflorez.com");
            v.add(p);
            try {
               File archivo = new File("archivo.txt");
               FileOutputStream fos = new FileOutputStream(archivo);
               ObjectOutputStream oos = new ObjectOutputStream(fos);
               oos.writeObject(v);
            } catch (FileNotFoundException e) {
               e.printStackTrace();
            } catch (IOException e) {
               e.printStackTrace();
            }
      }
}
```

Archivo "archivo.txt"


```
¬í sr java.util.VectorÙ—}[€;¯ ______________________________ I
capacityIncrementI elementCount[
elementDatat [Ljava/lang/Object;xp ________________ur [
Ljava.lang.Object; ÎXŸs)l  xp
sr Personaõ¬x"ôvÉ
I idL apellidot Ljava/lang/String;L correoq ~ L nombreq ~ xp   t
Florezt hf@hectorflorez.comt Hectorsq ~    t    Fernandezt af@
hectorflorez.comt Arturosq ~______t Valdezt jv@hectorflorez.comt
____________________________________________________________
Juanppppppx
```


El contenido del archivo "*archivo.txt*" es como se muestra en el cuadro anterior. El contenido de este archivo es diferente al del anterior, debido a que este archivo almacena solo un objeto que es instancia de la clase *Vector*.

7.3.2 Lectura en archivo serializable

Para realizar la lectura en un archivo serializable se puede hacer mediante el uso de las clases *File*, *FileInputStream* y *ObjectInputStream*.

La clase *ObjectInputStream* requiere un objeto instancia de la clase *FileInputStream* para poder abrir el archivo. A su vez opcionalmente, la clase *FileInputStream* puede requerir un objeto instancia de la clase *File*. Si no se desea usar un objeto de la clase *File*, al crear el objeto de la clase *FileInputStream*, se debe incluir la ruta del archivo.

Es importante tener en cuenta que si se intenta abrir un archivo con la clase *ObjectInputStream* y este no existe, entonces se presenta una excepción controlada por la clase *FileNotFoundException*.

En la lectura, se leer del archivo objetos, lo cual hace necesario realizar el "*casting*" correspondiente.

Con base en la clase *Persona* que implementa serializable, se hace la siguiente implementación.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class EjemplosArchivos {

   public static void main(String[] args) {
       Persona p;
       try {
          File archivo = new File("archivo.txt");
          FileInputStream fis = new FileInputStream(archivo);
          ObjectInputStream ois = new ObjectInputStream(fis);
          p=(Persona)ois.readObject();
          System.out.println(p.toString());
          p=(Persona)ois.readObject();
          System.out.println(p.toString());
          p=(Persona)ois.readObject();
          System.out.println(p.toString());
       } catch (FileNotFoundException e) {
          e.printStackTrace();
       } catch (IOException e) {
          e.printStackTrace();
       } catch (ClassNotFoundException e) {
          e.printStackTrace();
       }
   }
}
```

Salida estándar
id:1; Nombre: Héctor; Apellido: Flórez; Correo: hf@hectorflorez.com id:2; Nombre: Arturo; Apellido: Fernández; Correo: af@hectorflorez.com id:3; Nombre: Juan; Apellido: Valdez; Correo: jv@hectorflorez.com

De la misma forma que en la escritura se ha replicado la línea de código que usa el método *readObject*. Con base en la solución planteada usando la clase *Vector* se puede realizar la implementación, realizando la lectura de un solo objeto. Esta modificación agrega robustez a la aplicación, debido a que solo lee un objeto *instancia* de la clase *Vector*, el cual puede contener diferentes objetos *instancias* de diferentes clases.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.Iterator;
import java.util.Vector;

public class EjemplosArchivos {

   public static void main(String[] args) {
       Persona p;
       Vector v = new Vector();
       try {
          File archivo = new File("archivo.txt");
          FileInputStream fis = new FileInputStream(archivo);
          ObjectInputStream ois = new ObjectInputStream(fis);
          v=(Vector)ois.readObject();
       } catch (FileNotFoundException e) {
          e.printStackTrace();
       } catch (IOException e) {
          e.printStackTrace();
       } catch (ClassNotFoundException e) {
          e.printStackTrace();
       }
       Iterator iterator = v.iterator();
       while(iterator.hasNext()){
          p=(Persona)iterator.next();
          System.out.println(p.toString());
       }
   }
}
```

Salida estándar

`id:1; Nombre: Héctor; Apellido: Flórez; Correo: hf@hectorflorez.com`
`id:2; Nombre: Arturo; Apellido: Fernández; Correo: af@hectorflorez.com`
`id:3; Nombre: Juan; Apellido: Valdez; Correo: jv@hectorflorez.com`

En los dos casos, la salida es exactamente igual porque la información que se imprime en la salida estándar es proporcionada por la clase *Persona*, la cual no ha recibido ninguna modificación.

7.4 Archivos *Properties*

Un archivo "*properties*" es un archivo secuencial que es usado exclusivamente para lectura de información de propiedades para una aplicación. Este archivo se caracteriza porque en cada línea debe haber un nombre de atributo seguido del carácter "=" seguido del valor del atributo. Además, este archivo siempre debe tener la extensión "*.properties*". El siguiente es un ejemplo de archivo *properties*.

Archivo "propiedades*.properties*"

`Propiedad1=valor1`
`Propiedad2=valor2`
`Propiedad3=valor3`
`Propiedad4=valor4`

El valor de un atributo en un archivo *properties* siempre se lee de tipo *String*. Para abrir un archivo *properties* se debe crear un objeto de la clase *File* con el nombre del archivo. Con este objeto se crea un objeto de la clase *FileInputStream*. Finalmente, se crea un objeto de la clase *Properties* y usando el método *load* se carga el archivo. Para leer los valores del archivo se debe utilizar el método *getProperty,* el cual recibe por parámetro el nombre del atributo. Un ejemplo de la lectura de un archivo *properties* es la siguiente:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
import java.io.IOException;
import java.util.Properties;

public class EjemplosArchivos {

   public static void main(String[] args) {
       File archivo = new File("propiedades.properties");
       try {
          FileInputStream fis = new FileInputStream(archivo);
          Properties archivoPropiedades = new Properties();
          archivoPropiedades.load(fis);
          for(int i=1; i<=4; i++){
              String dato = archivoPropiedades.getProperty(
              "Propiedad"+i);
              System.out.println(dato);
          }
       } catch (FileNotFoundException e) {
          e.printStackTrace();
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

Salida estándar
valor1 valor2 valor3 valor4

7.5 Ejercicios propuestos

1. Realice una aplicación que permita capturar 10 números por entrada estándar y los almacene en un archivo de texto plano.
2. Realice una aplicación que permita leer un archivo que contenga números y calcule el promedio de dichos números.
3. Realice una aplicación que permita leer y escribir múltiples objetos serializables en un archivo.

CAPÍTULO 8

Herencia y polimorfismo

8.1 Herencia

La herencia es el concepto que permite se puedan definir nuevas clases basadas en clases existentes, con el fin de reutilizar el código previamente desarrollado, generando una jerarquía de clases dentro de la aplicación. Entonces, si una clase se deriva de otra, esta hereda sus atributos y métodos. La clase derivada puede añadir nuevos atributos y métodos y/o redefinir los atributos y métodos heredados. Para que un atributo y método puedan ser heredados es necesario que su visibilidad sea "*protected*".

En Java, a diferencia de otros lenguajes orientados a objetos, una clase solo puede derivar de una única clase, con lo cual, no es posible realizar herencia múltiple con base en clases. Sin embargo, es posible "*simular*" la herencia múltiple con base en las interfaces.

Un ejemplo del concepto de herencia puede ser considerando, los miembros de una institución de educación. La institución está conformada por personas, pero cada persona tiene un rol dentro de la institución, que podría ser de empleado, estudiante o egresado. Así mismo, de empleado se podría derivar la clasificación, académico y administrativo. De académico se puede derivar, decano, coordinador y docente. De administrativo se puede derivar de acuerdo a la cantidad de departamentos de la institución.

La representación de herencia del caso anteriormente expuesto en lenguaje de modelado es la siguiente:

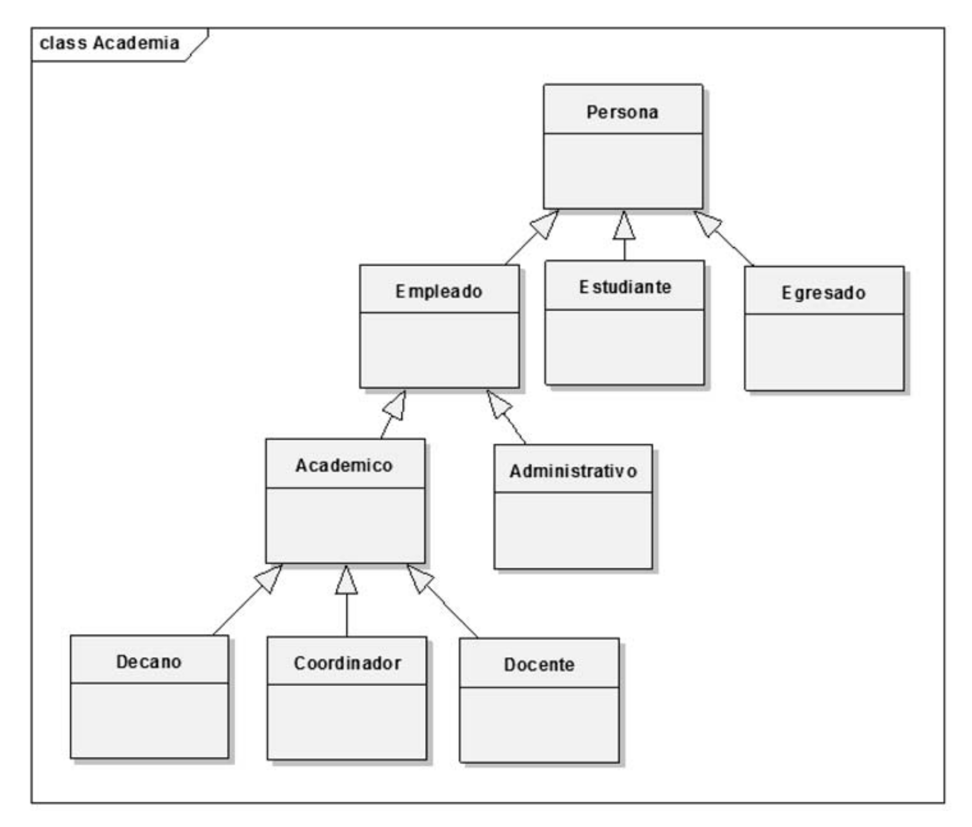


Figura 7. Jerarquía de herencia de personal académico

Otro ejemplo del concepto de herencia es el de figuras geométricas. Se puede considerar una clase denominada *FiguraGeometrica*, del cual heredan las clases *Cuadrado*, *Círculo*, Triángulo y Rectángulo.

En este caso, la clase *FiguraGeometrica*, poseería un atributo que puede ser llamado *valor1*. Este atributo es heredado por las clases *Cuadrado, Círculo, Triángulo* y *Rectángulo.* Sin embargo, las clases *Rectángulo* y *Triángulo* requieren dos valores. Esto indica que deben incluirse como atributos de cada una de estas clases. Por otro lado, la clase *FiguraGeometrica* puede implementar los métodos consultores y modificadores, los que podrán ser usados por cada una de las clases que la heredan.

Así mismo, de la clase *Cuadrado*, es posible heredar la clase *Cubo*. De la clase *Triángulo* es posible heredar la clase *Pirámide* y *Cono*.

De la clase Círculo es posible heredar la clase *Esfera* y *Cilindro*. La representación de herencia del caso anteriormente expuesto, en lenguaje de modelado, es la siguiente:

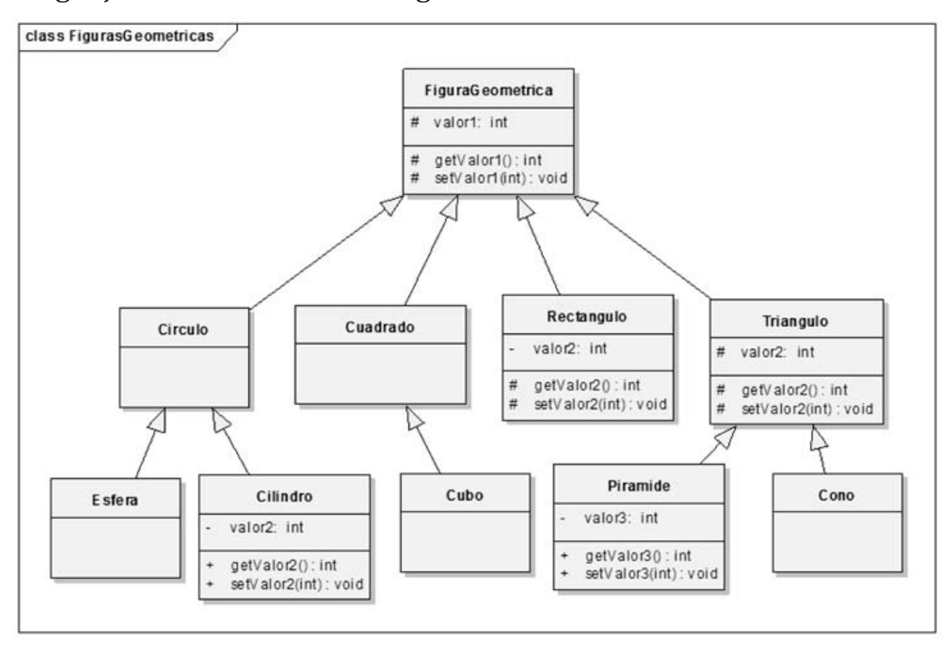


Figura 8. Jerarquía de herencia de figuras geométricas.

8.1.1 Sentencia *extends*

La sentencia "*extends*" permite implementar el concepto de herencia. Se incluye para que una clase herede de otra clase. Por ejemplo, en el caso de jerarquía de herencia de personal académico, debe existir una clase *persona* y una clase *estudiante*. Al implementar la clase *estudiante* se le debe incluir la sentencia *extends* para que herede de la clase *persona*. La sintaxis es la siguiente:

```
public class Persona{
   ...
}

public class Estudiante extends Persona{
   ...
}
```

Una vez incluida la sentencia *extends*, la clase *Estudiante* tiene acceso a atributos y métodos protegidos de la clase *Persona*.

8.1.2 Sentencia *super*

La sentencia "*super*" es utilizada para acceder a métodos implementados en la clase superior en el concepto de herencia. Esta sentencia es comúnmente utilizada para acceder al constructor de la clase superior desde el constructor de la clase inferior. Por ejemplo, en el caso de jerarquía de herencia de personal académico debe existir una clase *persona* con atributos como identificación, nombre, apellido y correo, y una clase *Estudiante* que puede acceder a estos atributos pero que adicionalmente, tiene atributos como código y facultad. Al implementar el constructor de la clase *estudiante* para asignar los valores de los atributos, se puede hacer un llamado al constructor de la clase *persona* enviándole los parámetros definidos en dicha clase. La sintaxis es la siguiente:

```
public class Persona{

   protected int id
   protected String nombre
   protected String apellido
   protected String correo

   public Persona(int id, String nombre, String apellido,
   String correo){
       this.id=id;
       this.nombre=nombre;
       this.apellido=apellido;
       this.correo=correo;
   }
}

public class Estudiante extends Persona{

   private int codigo
   private String facultad

   public Estudiante(int id, String nombre, String apellido,
   String correo, int codigo, String facultad){
```

```
        super(id, nombre, apellido, correo);
        this.codigo=codigo;
        this.facultad=facultad;
    }
}
```

En el ejemplo anterior, el constructor de la clase *estudiante*, hace un llamado al constructor de la clase *persona* asignando los valores a los atributos allí definidos.

8.1.3 Sobre-escritura de métodos

La sobre-escritura de métodos es una característica que se presenta en el concepto de herencia, que consiste en implementar un método en la clase superior e inferior en la jerarquía de herencia. Por ejemplo, considerando las clases *cuadrado* y *cubo* de la jerarquía de figuras geométricas que se presenta en la Figura 9, es posible crear un método *getArea,* tanto para la clase *cuadrado* como para la clase *cubo*. Entonces, si se crea una referencia de la clase *cuadrado*, dependiendo de la instancia del objeto que se crea que puede ser de *cuadrado* o *cubo*, se accede al método implementado en *cuadrado* o en *cubo*, respectivamente.

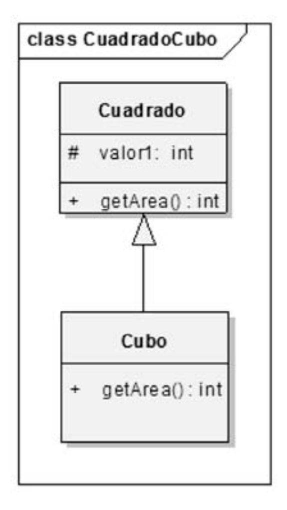


Figura 9. Jerarquía de herencia de Cuadrado y Cubo

La implementación de esta jerarquía es la siguiente:

Clase *Cuadrado*

```
package figurasGeometricas;

public class Cuadrado {
   protected int valor1;

   public Cuadrado(double valor1) {
       this.valor1=valor1;
   }

   public double getArea() {
       return Math.pow(this.valor1, 2);
   }
}
```

Clase *Cubo*

```
package figurasGeometricas;

public class Cubo extends Cuadrado {

   public Cubo(double valor1) {
       super(valor1);
   }

   public double getArea() {
       return Math.pow(this.valor1, 3);
   }
}
```

En la implementación anterior, sobre escribe el método *getArea* debido a que el área del cuadrado es diferente al área del cubo. Java identifica a cual método sobre-escrito debe acceder en tiempo de ejecución.

8.1.4 Clases abstractas

Una clase abstracta es aquella que no puede ser instanciada, es decir, no se pueden crear objetos de esta clase. Se usa para permitir que

otras clases hereden de esta proporcionando atributos y métodos que son comunes de las clases heredadas. La sintaxis para la creación de una clase abstracta es la siguiente:

```
public abstract class FiguraGeometrica {
   ...
}
```

Una clase abstracta puede contener atributos y métodos. Sin embargo, adicionalmente puede contener métodos abstractos, los cuales son definidos pero no implementados. Su finalidad es que las clases que heredan de la clase abstracta, implementen de forma obligatoria dichos métodos abstractos.

Con base en la jerarquía de herencia de figuras geométricas, la implementación de la clase *FiguraGeometrica* es la siguiente:

```
package figurasGeometricas;

public abstract class FiguraGeometrica {

   protected double valor1;

   public FiguraGeometrica(double valor1) {
       super();
       this.valor1 = valor1;
   }

   public double getValor1() {
       return valor1;
   }

   public void setValor1(double valor1) {
       this.valor1 = valor1;
   }

   public abstract double getArea();
   public abstract double getPerimetro();
}
```

De esta forma, las clases que hereden de la clase *FiguraGeometrica* pueden acceder al atributo *valor1*, a los métodos *getValor1* y *setValor1* y deben implementar los métodos *getArea* y *getPerimetro.*

Cuando se implementa una clase abstracta es importante tener en cuenta que, la clase no debe ser instanciada. Por ejemplo, en el caso de *FiguraGeometrica*, no se debe crear un objeto instancia de esta clase debido a que *FiguraGeometrica* en la realidad, actúa como una generalización, es decir, para realizar cálculos sobre una figura es necesario determinar a qué figura se hace referencia como por ejemplo, un cuadrado o un triángulo. Por otro lado, al definir un método abstracto es necesario identificar qué servicios deben obligatoriamente, implementar las clases que hereden de la clase abstracta. Por ejemplo, cualquier figura como cuadrado o triángulo tiene área y tiene perímetro. Por consiguiente, estos servicios deben ser implementados mediante la definición de un método abstracto.

De esta manera la implementación de las clases que heredan de *FiguraGeometrica* es la siguiente:

Clase *Cuadrado*

```
package figurasGeometricas;

public class Cuadrado extends FiguraGeometrica {

   public Cuadrado(double valor1) {
       super(valor1);
   }

   @Override
   public double getArea() {
       return Math.pow(this.valor1, 2);
   }

   @Override
   public double getPerimetro() {
       return this.valor1*4;
   }
}
```

Clase *Círculo*

```
package figurasGeometricas;

public class Circulo extends FiguraGeometrica {
```

```
    public Circulo(double valor1) {
        super(valor1);
    }

    @Override
    public double getArea() {
        return Math.PI*Math.pow(this.valor1, 2);
    }

    @Override
    public double getPerimetro() {
        return Math.PI*this.valor1;
    }
}
```

Clase *Triángulo*

```
package figurasGeometricas;

public class Triangulo extends FiguraGeometrica {
    private double valor2;

    public Triangulo(double valor1, double valor2) {
        super(valor1);
        this.valor2 = valor2;
    }

    public double getValor2() {
        return valor2;
    }

    public void setValor2(double valor2) {
        this.valor2 = valor2;
    }

    @Override
    public double getArea() {
        return (this.valor1*this.valor2)/2;
    }

    @Override
    public double getPerimetro() {
        return this.valor1 + (2 * Math.sqrt((Math.pow(
        this.valor1, 2)+Math.pow(this.valor2, 2))));
    }
}
```

Clase *Rectángulo*

```
package figurasGeometricas;

public class Rectangulo extends FiguraGeometrica {
   private double valor2;

   public Rectangulo(double valor1, double valor2) {
       super(valor1);
       this.valor2 = valor2;
   }

   public double getValor2() {
       return valor2;
   }

   public void setValor2(double valor2) {
       this.valor2 = valor2;
   }

   @Override
   public double getArea() {
       return this.valor1*this.valor2;
   }

   @Override
   public double getPerimetro() {
       return 2*this.valor1 + 2*this.valor2;
   }
}
```

8.1.5 Interfaces

Una interfaz es un tipo especial de clase que permite realizar un conjunto de declaraciones de métodos sin implementación. En una interfaz también se pueden definir constantes que, son implícitamente *public*, *static* y *finaly;* deben siempre inicializarse en la declaración. Para que una clase use las definiciones de una interfaz, dicha clase debe incluir la sentencia "*implements*" la cual indica que implementa la interfaz. La sintaxis es la siguiente:

```
public interface MiInterfaz {
   ...
}

public class MiClase implements MiInterfaz {
   ...
}
```

El objetivo de los métodos declarados en una interfaz es definir un tipo de conducta para las clases que implementan dicha interfaz. Todas las clases que colocan en funcionamiento una determinada interfaz, están obligadas a proporcionar una implementación de los métodos declarados en la interfaz adquiriendo un comportamiento.

Una clase puede implementar una o varias interfaces. Para indicar que una clase implementa más de una interfaz, se ponen los nombres de las interfaces separados por comas, posterior a incluir la sentencia "*implements*". La sintaxis es la siguiente:

```
public class MiClase implements MiInterfaz1, MiInterfaz2,
MiInterfazN  {
   ...
}
```

Un ejemplo del uso de interfaces con base en el modelo de herencia de figuras geométricas es el siguiente. En este ejemplo, se plantea la posibilidad de dibujar una figura como círculo o rectángulo. Para ello, se crea una interfaz denominada *FiguraDibujable* que contiene métodos que definen comportamiento de dibujo de la figura. Entonces se puede hacer una clase que herede de *Círculo* e implemente *FiguraDibujable* y una clase que herede de *Rectángulo* e implemente *FiguraDibujable*.

Clase *FiguraGeometrica*

```
package figurasGeometricas;

public abstract class FiguraGeometrica {

   protected double valor1;

   public FiguraGeometrica(double valor1) {
       super();
       this.valor1 = valor1;
   }

   public double getValor1() {
       return valor1;
   }
```

```
    public void setValor1(double valor1) {
        this.valor1 = valor1;
    }

    public abstract double getArea();
    public abstract double getPerimetro();
}
```

Interfaz *FiguraDibujable*

```
package figurasGeometricas;

import java.awt.Graphics;

public interface FiguraDibujable {
    public void setCoordenadas(int x, int y);
    public void dibujar2D(Graphics g);
}
```

Clase *Círculo*

```
package figurasGeometricas;

public class Circulo extends FiguraGeometrica {

    public Circulo(double valor1) {
        super(valor1);
    }

    @Override
    public double getArea() {
        return Math.PI*Math.pow(this.valor1, 2);
    }

    @Override
    public double getPerimetro() {
        return Math.PI*this.valor1;
    }
}
```

Clase *CírculoDibujable*

```
package figurasGeometricas;
import java.awt.Graphics;
```

```
public class CirculoDibujable extends Circulo implements
FiguraDibujable {
   private int x;
   private int y;

   public CirculoDibujable(double valor1, int x, int y) {
       super(valor1);
       this.x = x;
       this.y = y;
   }

   @Override
   public void setCoordenadas(int x, int y) {
       this.x=x;
       this.y=y;
   }

   @Override
   public void dibujar2D(Graphics g) {
       g.drawOval(this.x, this.y, (int)this.valor1, (
       int)this.valor1);
    }
}
```

Clase *Rectángulo*

```
package figurasGeometricas;

public class Rectangulo extends FiguraGeometrica {
   protected double valor2;

   public Rectangulo(double valor1, double valor2) {
       super(valor1);
       this.valor2 = valor2;
   }

   public double getValor2() {
       return valor2;
   }

   public void setValor2(double valor2) {
       this.valor2 = valor2;
   }

   @Override
   public double getArea() {
       return this.valor1*this.valor2;
   }
```

```
   @Override
   public double getPerimetro() {
       return 2*this.valor1 + 2*this.valor2;
   }
}
```

Clase *RectánguloDibujable*

```
package figurasGeometricas;

import java.awt.Graphics;

public class RectanguloDibujable extends Rectangulo implements
FiguraDibujable {
   private int x;
   private int y;

   public RectanguloDibujable(double valor1, double valor2,
   int x, int y) {
       super(valor1, valor2);
       this.x = x;
       this.y = y;
   }

   @Override
   public void setCoordenadas(int x, int y) {
       this.x=x;
       this.y=y;
   }

   @Override
   public void dibujar2D(Graphics g) {
       g.drawRect(this.x, this.y, (int)this.valor1, (
       int)this.valor2);
   }
}
```

8.2 Polimorfismo

El polimorfismo es la característica de la programación orientada a objetos, que permite modificar la instancia de un objeto en tiempo de ejecución basado en una jerarquía de herencia. De esta forma, es posible generar una relación de vinculación denominada "*binding*". El polimorfismo se puede realizar con clases superiores normales, abstractas e interfaces.

El objetivo del polimorfismo consiste en poder acceder a diferentes servicios en tiempo de ejecución sin necesidad de implementar diferentes referencias a objetos. Esta característica provee una gran flexibilidad en el proceso de desarrollo y ejecución de la aplicación.

Por ejemplo, considerando la jerarquía de herencia de *Figuras Geométricas* de la siguiente figura es posible hacer uso del concepto de polimorfismo.

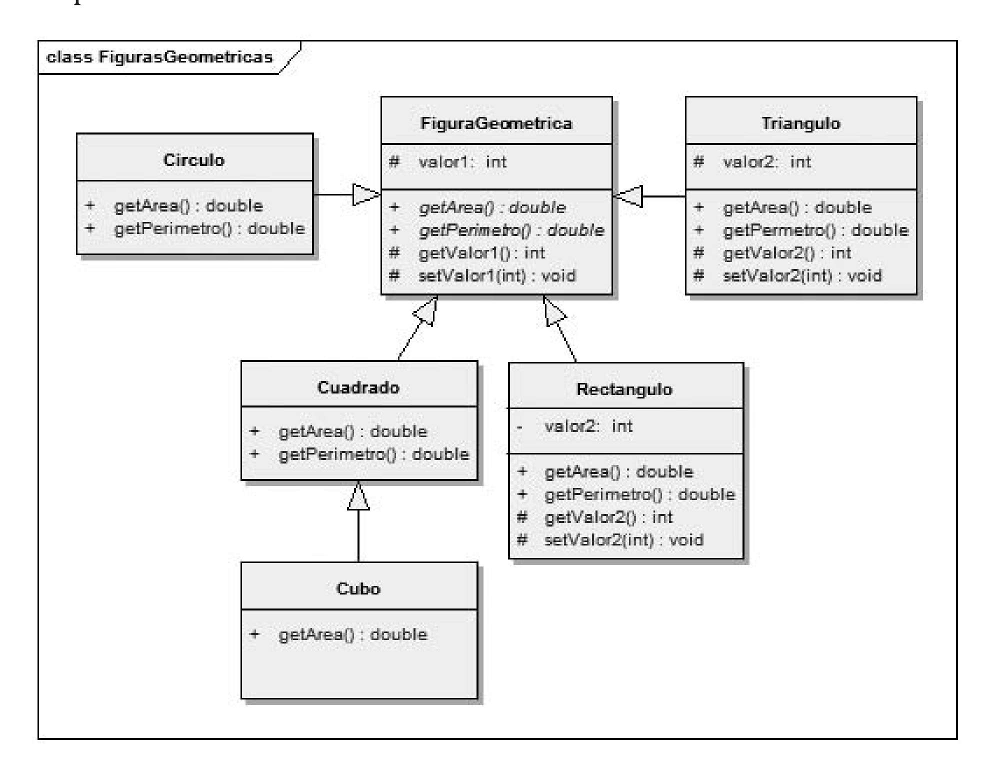


Figura 10. Jerarquía de herencia de Figuras Geométricas

Con base en la Figura 10 se puede crear una referencia de la clase *FiguraGeometrica* de la siguiente forma:

```
FiguraGeometrica figura;
```

A la referencia *figura* se le puede generar instancia de cualquiera de las clases que derivan de ella de la siguiente forma:

```
figura = new Circulo(5);
figura = new Cuadrado(5);
figura = new Rectangulo(5,2);
figura = new Triangulo(5,2);
figura = new Cubo(5);
```

En la primera línea, el objeto *figura* tiene la forma de círculo, de tal forma que si se accede al método *getArea*, se ejecuta el método implementado en círculo.

En la segunda línea, el objeto *figura* tiene la forma de cuadrado, de tal forma que si se accede al método *getArea*, se ejecuta el método implementado en cuadrado.

En la tercer línea, el objeto *figura* tiene la forma de rectángulo, de tal forma que si se accede al método *getArea*, se ejecuta el método implementado en rectángulo.

En la cuarta línea, el objeto *figura* tiene la forma de triángulo, de tal forma que si se accede al método *getArea*, se ejecuta el método implementado en triángulo.

En la quinta línea, el objeto *figura* tiene la forma de cubo, de tal forma que si se accede al método *getArea*, se ejecuta el método implementado en cubo.

La siguiente implementación del modelo presentado demuestra en tiempo de ejecución el concepto de polimorfismo.

Clase *FiguraGeometrica*

```
package figurasGeometricas;

public abstract class FiguraGeometrica {

   protected double valor1;

   public FiguraGeometrica(double valor1) {
       super();
       this.valor1 = valor1;
   }
```

```
    public double getValor1() {
        return valor1;
    }

    public void setValor1(double valor1) {
        this.valor1 = valor1;
    }

    public abstract double getArea();
    public abstract double getPerimetro();
}
```

Clase *Círculo*

```
package figurasGeometricas;

public class Circulo extends FiguraGeometrica {

    public Circulo(double valor1) {
        super(valor1);
    }

    @Override
    public double getArea() {
        return Math.PI*Math.pow(this.valor1, 2);
    }

    @Override
    public double getPerimetro() {
        return Math.PI*this.valor1;
    }
}
```

Clase *Cuadrado*

```
package figurasGeometricas;

public class Cuadrado extends FiguraGeometrica {

    public Cuadrado(double valor1) {
        super(valor1);
    }

    @Override
    public double getArea() {
```

```
        return Math.pow(this.valor1, 2);
    }

    @Override
    public double getPerimetro() {
        return this.valor1*4;
    }
}
```

Clase *Rectángulo*

```
package figurasGeometricas;

public class Rectangulo extends FiguraGeometrica {
    protected double valor2;

    public Rectangulo(double valor1, double valor2) {
        super(valor1);
        this.valor2 = valor2;
    }

    public double getValor2() {
        return valor2;
    }

    public void setValor2(double valor2) {
        this.valor2 = valor2;
    }

    @Override
    public double getArea() {
        return this.valor1*this.valor2;
    }

    @Override
    public double getPerimetro() {
        return 2*this.valor1 + 2*this.valor2;
    }
}
```

Clase *Triángulo*

```
package figurasGeometricas;
public class Triangulo extends FiguraGeometrica {
    private double valor2;
```

```
    public Triangulo(double valor1, double valor2) {
        super(valor1);
        this.valor2 = valor2;
    }

    public double getValor2() {
        return valor2;
    }

    public void setValor2(double valor2) {
        this.valor2 = valor2;
    }

    @Override
    public double getArea() {
        return (this.valor1*this.valor2)/2;
    }

    @Override
    public double getPerimetro() {
        return this.valor1 + (2 * Math.sqrt((
        Math.pow(this.valor1, 2)+Math.pow(this.valor2, 2))));
    }
}
```

Clase *Cubo*

```
package figurasGeometricas;

public class Cubo extends Cuadrado {

    public Cubo(double valor1) {
        super(valor1);
    }

    public double getArea() {
        return Math.pow(this.valor1, 3);
    }
}
```

Clase *Principal*

```
package figurasGeometricas;

public class Principal {
```

```
    public static void main(String[] args) {
        FiguraGeometrica figura;
        figura = new Circulo(5);
        System.out.println(figura.getClass());
        System.out.println("Area: "+figura.getArea());
        System.out.println("Perimetro: "+figura.getPerimetro());
        figura = new Cuadrado(5);
        System.out.println(figura.getClass());
        System.out.println("Area: "+figura.getArea());
        System.out.println("Perimetro: "+figura.getPerimetro());
        figura = new Rectangulo(5,2);
        System.out.println(figura.getClass());
        System.out.println("Area: "+figura.getArea());
        System.out.println("Perimetro: "+figura.getPerimetro());
        figura = new Triangulo(5,2);
        System.out.println(figura.getClass());
        System.out.println("Area: "+figura.getArea());
        System.out.println("Perimetro: "+figura.getPerimetro());
        figura = new Cubo(5);
        System.out.println(figura.getClass());
        System.out.println("Area: "+figura.getArea());
        System.out.println("Perimetro: "+figura.getPerimetro());
    }
}
```

Salida estándar

```
class FigurasGeometricas.Circulo
Area: 78.53981633974483
Perimetro: 15.707963267948966
class FigurasGeometricas.Cuadrado
Area: 25.0
Perimetro: 20.0
class FigurasGeometricas.Rectangulo
Area: 10.0
Perimetro: 14.0
class FigurasGeometricas.Triangulo
Area: 5.0
Perimetro: 15.770329614269007
class FigurasGeometricas.Cubo
Area: 125.0
Perimetro: 20.0
```

En la implementación de la clase *Principal* se puede apreciar que el objeto *figura* cambia de forma cada vez que se hace una nueva

instancia. Así mismo, en cada instancia se accede a la implementación del método de la clase instanciada.

El último caso realiza la instancia de la clase *Cubo*, en donde esta clase no tiene implementado el método *getPerimetro*. Para este caso, en el llamado a este método, se accede al método *getPerimetro* implementado en la clase superior que es *Cuadrado*.

8.3 Ejercicios propuestos

1. Implemente una aplicación que a partir de un modelo de herencia represente animales con las siguientes clasificaciones.
    a. Mamífero
        i. Canino
            1. Perro
            2. Lobo
        ii. Felino
            1. Gato
            2. Tigre
    b. Ovíparo
        i. Ave
            1. Águila
            2. Paloma
        ii. Pez
            1. Trucha
            2. Salmón

2. Con base en el ejercicio anterior implemente una interfaz que contenga métodos de comportamiento de los animales descritos.

CAPÍTULO 9

Documentación con *Javadoc*

Javadoc genera documentación *API* en formato *HTML*, para el paquete especificado o para las clases individuales en Java. También se puede indicar una serie de paquetes o clases Java con los que se desea generar la documentación. *Javadoc* genera un archivo con extensión "***.html***" por cada clase ***.java*** y paquete que encuentra. También genera la jerarquía de clases en un archivo denominado *tree.html* y un índice con todos los miembros que ha detectado en un archivo denominado *AllNames.html.*

Para generar la documentación se debe utilizar la aplicación *Javadoc* que se encuentra en la ruta:

```
[jdk]/bin/javadoc [nombrePaquete]|[archivos.java]
```

En donde la ruta **[jdk]** representa la ubicación donde se encuentra instalada la máquina virtual de Java. El parámetro **[nombrePaquete]** hace referencia al paquete al que se le aplicara *Javadoc* y el parámetro **[archivo.java]** hace referencia a la clase a la que se le aplicará *Javadoc*.

Se puede configurar el contenido y el formato de salida que se va a generar con *Javadoc* a través de *doclets*. Un *doclet* es un programa Java escrito utilizando el *Java Doclet API*, que especifica el contenido y formato de la salida que ha de generar *Javadoc.* Se pueden escribir *doclets* para generar cualquier tipo de salida de texto, ya sea *HTML, SGML, RTF* o *MIF*.

Cuando no se indica ningún *doclet* específico en la línea de comandos, *Javadoc* utilizará el *doclet* estándar que genera una salida *HTML*.

9.1 Documentación de código fuente

Para generar el *Javadoc* no es necesario hacer comentarios a todas las líneas de código. Lo único estrictamente necesario es comentar la clase y cada uno de sus métodos. Como *Javadoc* comprueba automáticamente clases, interfaces, métodos y atributos; se puede añadir documentación adicional a través de comentarios de documentación en el código fuente.

Javadoc reconoce marcas especiales cuando recorre los comentarios de documentación. Estas marcas permiten autogenerar una documentación completa y bien formateada del *API* a partir del código fuente. Las marcas comienzan siempre con el signo @. Estas marcas deben situarse al principio de la línea y todas las marcas con el mismo nombre deben agruparse juntas dentro del comentario de documentación. Solamente se indicarán las más representativas agrupadas por clases, métodos y atributos.

Para documentar clases e interfaces se cuenta con las siguientes etiquetas:

```
@version 1.0
```

Incluye información de versión de la clase. Un comentario de documentación puede incluir más de una marca @*version.*

```
@author Hector Florez
```

Incluye información de versión de la clase. Un comentario de documentación puede incluir más de una marca @*author.*

```
@see java.lang.String
```

Incluye un enlace a la documentación de una clase a la clase en la zona "*See Also*". Un comentario de documentación puede incluir más de una marca *@see.*

```
@see java.lang.String#valueOf
```

El carácter # separa el nombre de una clase del nombre de uno de sus atributos o métodos. Para documentar métodos se cuenta con las siguientes etiquetas:

```
@param parametro descripcion
```

Incluye información de parámetro del método. Cada método debe incluir tantas etiquetas como parámetros.

```
@return descripcion
```

Incluye información de retorno del método.

```
@exception nombredelaclase de Excepcion descripcion
```

Incluye información de manejo de excepción que puede ser lanzada por el método. La excepción estará enlazada con su clase en la documentación.

9.2 Resultados de Javadoc

Considerando la implementación de figuras geométricas presentadas en el capítulo anterior se puede generar la siguiente documentación:

Clase *FiguraGeometrica*

```
package figurasGeometricas;

/**
 * Clase abstracta que hereda a Circulo, Cuadrado,
 Triangulo y Rectangulo
 * @version 1.0
```

```
 * @author Hector Florez
 */
public abstract class FiguraGeometrica {

   protected double valor1;

   /**
    * Constructor. Asigna un valor a la figura geometrica
    * @param valor1 Recibe valor del atributo a traves del
    constructor
    */
   public FiguraGeometrica(double valor1) {
       super();
       this.valor1 = valor1;
   }

   /**
    * Metodo consultor del atributo valor1
    * @return Retorna el valor del atributo a traves del metodo
    consultor
    */
   public double getValor1() {
       return valor1;
   }

   /**
    * Metodo modificador del atributo valor1
    * @param valor1 Recibe valor del atributo a traves del
    metodo modificador
    */
   public void setValor1(double valor1) {
       this.valor1 = valor1;
   }

   /**
    * Metodo abstracto que define el servicio calcular area
    * @return Retorna el area de la figura
    */
   public abstract double getArea();

   /**
    * Metodo abstracto que define el servicio calcular perimetro
    * @return Retorna el perimetro de la figura
    */
   public abstract double getPerimetro();
}
```

Clase *Círculo*

```
package figurasGeometricas;

/**
 * Clase que provee servicios para el TAD Circulo
 * @version 1.0
 * @author Hector Florez
 * @see Math
 */
public class Circulo extends FiguraGeometrica {

   /**
    * Constructor. Asigna un valor al circulo.
    * Llama al constructor de FiguraGeometrica
    * @param valor1 Recibe valor del atributo a traves del
    constructor
    */
    public Circulo(double valor1) {
       super(valor1);
   }

   /**
    * Metodo sobre-escrito que calcula el area del circulo
    * @return Retorna el area del circulo
    */
   @Override
   public double getArea() {
       return Math.PI*Math.pow(this.valor1, 2);
   }

   /**
    * Metodo sobre-escrito que calcula el perimetro del circulo
    * @return Retorna el perimetro del circulo
    */
   @Override
   public double getPerimetro() {
       return Math.PI*this.valor1;
   }
}
```

Documentando todas las clases del paquete *FigurasGeometricas*, al aplicar *Javadoc*, se obtienen los siguientes resultados:

Package Class Use **Tree** Deprecated Index Help
PREV NEXT FRAMES NO FRAMES All Classes

Hierarchy For All Packages

Package Hierarchies:
FigurasGeometricas

Class Hierarchy

- java.lang.Object
  - FigurasGeometricas.**FiguraGeometrica**
    - FigurasGeometricas.**Circulo**
      - FigurasGeometricas.**CirculoDibujable** (implements FigurasGeometricas.FiguraDibujable)
    - FigurasGeometricas.**Cuadrado**
      - FigurasGeometricas.**Cubo**
    - FigurasGeometricas.**Rectangulo**
      - FigurasGeometricas.**RectanguloDibujable** (implements FigurasGeometricas.FiguraDibujable)
    - FigurasGeometricas.**Triangulo**
  - FigurasGeometricas.**Principal**

Interface Hierarchy

- FigurasGeometricas.**FiguraDibujable**



Figura 11. *Javadoc*. Jerarquía de paquetes

Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

FigurasGeometricas

Class FiguraGeometrica

```
java.lang.Object
  └ FigurasGeometricas.FiguraGeometrica
```

Direct Known Subclasses:

Circulo, Cuadrado, Rectangulo, Triangulo

```
public abstract class FiguraGeometrica
extends java.lang.Object
```

Clase abstracta que hereda a Circulo, Cuadrado, Triangulo y Rectangulo

Version:

1.0

Author:

Hector Florez

Constructor Summary
`FiguraGeometrica(double valor1)` Constructor.

Method Summary	
`abstract double`	`getArea()` Metodo abstracto que define el servicio calcular area
`abstract double`	`getPerimetro()` Metodo abstracto que define el servicio calcular perimetro
`double`	`getValor1()` Metodo consultor del atributo valor1
`void`	`setValor1(double valor1)` Metodo modificador del atributo valor1

Figura 12. *Javadoc*. Documentación de clase abstracta

Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

FigurasGeometricas
Class Circulo

```
java.lang.Object
  └ FigurasGeometricas.FiguraGeometrica
      └ FigurasGeometricas.Circulo
```

Direct Known Subclasses:
CirculoDibujable

```
public class Circulo
extends FiguraGeometrica
```

Clase que provee servicios para el TAD Circulo

Version:
1.0
Author:
Hector Florez
See Also:
Math

Constructor Summary
Circulo(double valor1) Constructor.

Method Summary	
double	getArea() Metodo sobreescrito que calcula el area del circulo
double	getPerimetro() Metodo sobreescrito que calcula el perimetro del circulo

Methods inherited from class FigurasGeometricas.FiguraGeometrica
getValor1, setValor1

Figura 13. *Javadoc*. Documentación de clase que hereda de clase abstracta

Constructor Detail

Circulo

```
public Circulo(double valor1)
```

Constructor. Asigna un valor al circulo. Llama al contructor de FiguraGeometrica

Parameters:
`valor1` - Recibe valor del atributo a traves del constructor

Method Detail

getArea

```
public double getArea()
```

Metodo sobreescrito que calcula el area del circulo

Specified by:
`getArea` in class `FiguraGeometrica`

Returns:
Retorna el area del circulo

---

getPerimetro

```
public double getPerimetro()
```

Metodo sobreescrito que calcula el perimetro del circulo

Specified by:
`getPerimetro` in class `FiguraGeometrica`

Returns:
Retorna el perimetro del circulo

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Figura 14. *Javadoc*. Documentación de métodos de clase

CAPÍTULO 10

Desarrollo orientado a arquitecturas

Desarrollar una aplicación basada en una arquitectura, permitirá a la aplicación tener características fundamentales del paradigma de orientación a objetos como son bajo acoplamiento, alta cohesión y escalabilidad.

Una arquitectura define un conjunto de capas. Cada capa se debe especializar en servicios enfocados a resolver algún comportamiento o característica de la aplicación. Por ejemplo, si se tiene una aplicación que tiene elementos de interfaz gráfica, las clases relacionadas con esta interfaz deben estar separadas por medio de una capa de las clases que tienen otro objetivo en la aplicación como es el almacenamiento de datos o comunicación de información.

10.1 Arquitectura de tres capas

La arquitectura de tres capas es una técnica en el desarrollo de aplicaciones de *software* que tiene como objetivo la separación de la lógica del negocio de la presentación y de la persistencia. Una de las principales ventajas se obtiene con el bajo acoplamiento de las aplicaciones debido a que esta característica, permite fácilmente realizar cambios en los servicios sin tener que revisar todos los componentes de la aplicación. Además, esta técnica permite distribuir el trabajo de los desarrolladores por niveles, en donde cada equipo de desarrollo puede hacer uso de los componentes desarrollados por

otro equipo sin necesidad de conocer el desarrollo, solo conociendo los resultados de los servicios.

La división en componentes reduce la complejidad, permite la reutilización de código agilizando el proceso de desarrollo del producto de *software*. Por otro lado, esta estrategia, permite hacer correcto uso de las metodologías de desarrollo de *software*. Particularmente, la metodología *RUP (Rational Unified Proccess)* se adapta cómodamente a la arquitectura, permitiendo facilidad en los procesos de desarrollo de la aplicación. La arquitectura de tres capas se basa en el siguiente modelo.

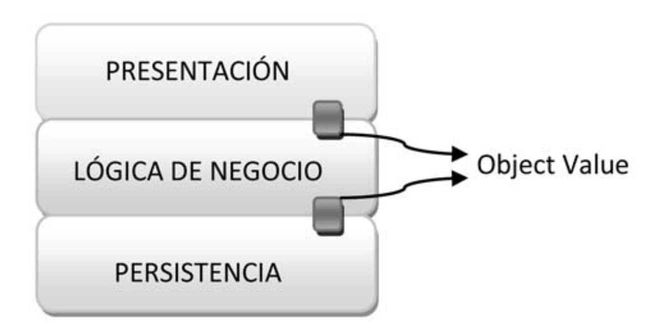


Figura 15. Arquitectura de tres capas

La capa de presentación exhibe la aplicación al usuario, le muestra la información y captura la información del usuario. Esta capa se comunica con la capa de lógica de negocio por medio de objetos que se denominan "*Object value*".

La capa de lógica de negocio es donde se desarrollan los algoritmos propios de la aplicación. En esta capa se implementa la lógica obtenida por el análisis de requerimientos del proyecto. Esta capa provee servicios a la capa de presentación, recibiendo como parámetros la información que el usuario entrega a la aplicación. Esta capa se comunica con la capa de persistencia por medio de objetos que se denominan "*Object value*".

La capa de persistencia es donde se almacenan los datos y se realiza las operaciones para manipular dichos datos. Estos datos pueden

encontrarse en cualquier tipo de sistema que almacene información en disco duro como archivos y bases de datos. Esta capa recibe solicitudes de almacenamiento o recuperación de información desde la capa de la lógica del negocio. Entonces esta capa es la encargada de abrir archivos o crear conexiones a bases de datos. Las conexiones a bases de datos deben realizarse a través de componentes denominados "*JDBC* (*Java Data Base Connector*)". La capa de persistencia usa el *JDBC* para manipular la información de la base de datos.

La arquitectura de tres capas tiene una característica adicional que es la facilidad de aplicación de patrones de desarrollo de *software*. Además, genera grandes beneficios para el proyecto porque permite realizar escalabilidad, portabilidad y usabilidad, entre otros.

La arquitectura de tres capas permite realizar grandes modificaciones a la aplicación de forma transparente para los procesos contenidos en ella. Por ejemplo, si se tiene una aplicación de escritorio y se desea modificar esa aplicación para *web*, basta con modificar la capa de presentación dejando intacta las otras capas. En este caso, la nueva capa de presentación hace uso de la capa de lógica accediendo a los servicios que esta capa presta. Otro ejemplo consiste en la posibilidad de cambiar de motor de base de datos. En este caso, bastaría con modificar la conexión y el *JDBC* respectivo sin tener que modificar todo el proyecto.

Se pueden plantear dos formas simples para aplicar una arquitectura de tres capas a un proyecto, dependiendo de su tamaño y complejidad.

La primera forma, consiste en crear tres paquetes que harán referencia a cada capa. En cada uno de estos paquetes se implementan las clases necesarias para proveer los servicios requeridos. En la capa de presentación debe estar ubicado el método *main*, con el fin de que la aplicación inicie su ejecución en esta capa. Esta implementación es válida para proyectos pequeños con pocos requerimientos.

La Figura 16 presenta la distribución de clases en un proyecto orientado a esta arquitectura utilizando la herramienta de desarrollo eclipse *IDE*.

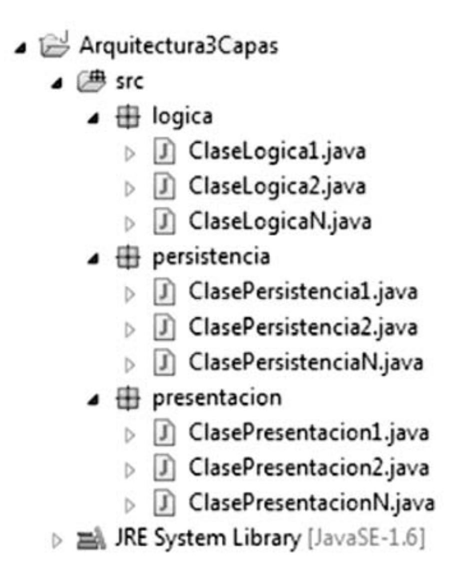


Figura 16. Implementación basada en paquetes de arquitectura de tres capas

La segunda forma consiste en crear tres proyectos, en donde cada proyecto hará referencia a cada capa. En cada uno de estos proyectos se implementan las clases necesarias para proveer los servicios requeridos. En el proyecto que hace referencia a la capa de presentación debe estar ubicado el método *main*, con el fin de que la aplicación inicie su ejecución en esta capa. Esta implementación es válida para cualquier tipo de proyectos.

Es necesario configurar el proyecto *Presentación* para que pueda acceder al proyecto *Lógica* y el proyecto *Lógica* para que pueda acceder al proyecto *Persistencia*. En eclipse *IDE* este procedimiento se realiza a través de la opción "*Configure Build Path*" que presenta una ventana de configuración como se muestra en la Figura 17.

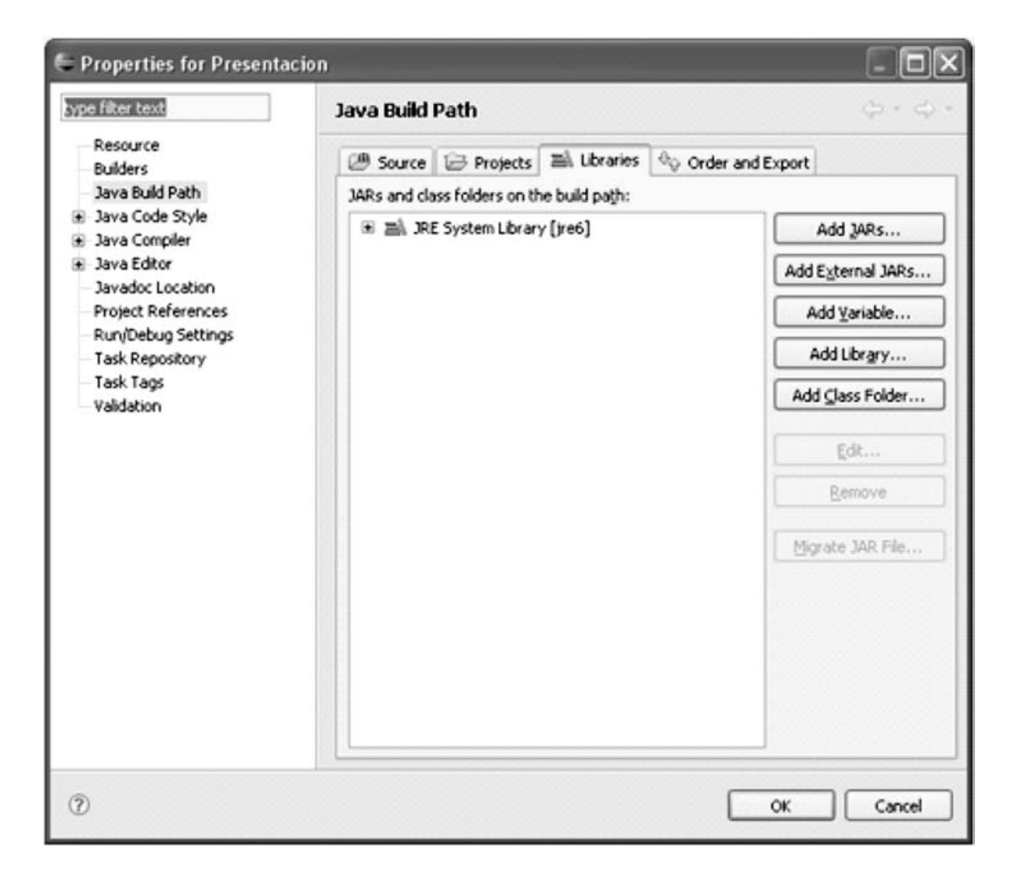


Figura 17. Configuración de *Build Path*

En la pestaña *Projects* se adiciona el proyecto correspondiente.

La Figura 18 presenta la distribución de clases en un proyecto orientado a esta arquitectura utilizando la herramienta de desarrollo eclipse *IDE*.

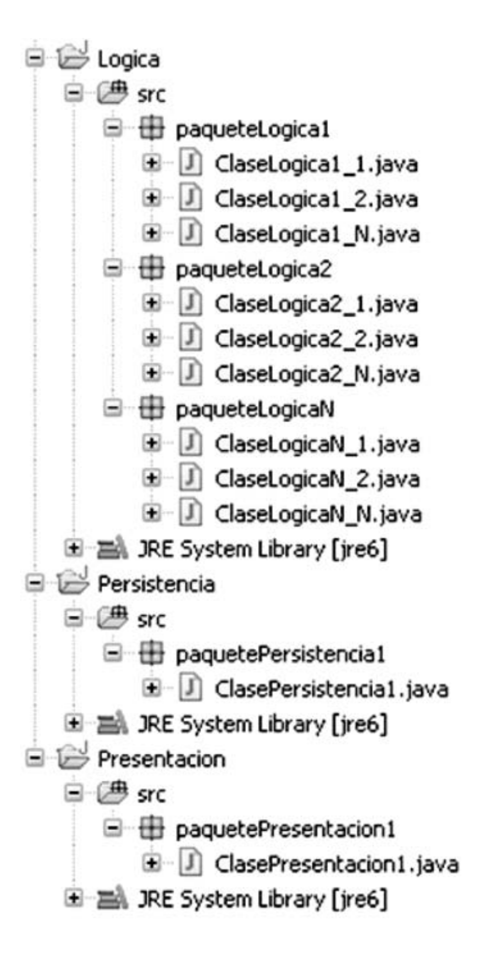


Figura 18. Implementación basada en proyectos de arquitectura de tres capas

10.2 Arquitectura multicapa

Una arquitectura multicapa se basa en la arquitectura de tres capas. Con frecuencia, cada capa de la arquitectura de tres capas se puede componer de varias capas. Se puede considerar las siguientes situaciones:

- Si se tiene una aplicación *web*, la capa de presentación, podría contener componentes para el uso de *AJAX* (*Asincronous Javascript And Xml*), de esta forma, la capa de presentación

se compondría de las clases que hacen parte de la lógica de presentación, más una capa adicional *AJAX* que contendría las clases para proveer estos servicios.

- Si se tiene una capa de lógica de negocio, en donde se requiere encriptación, se puede crear una nueva capa que contenga servicios de criptografía.
- Si se desea implementar el patrón *ORM* (*Object Relationship Mapping*) en la capa de persistencia, es posible dividir esta capa en una capa que contenga las clases necesarias para realizar la conexión a las bases de datos y otra capa que contenga clases de mapeo relacional de objetos.

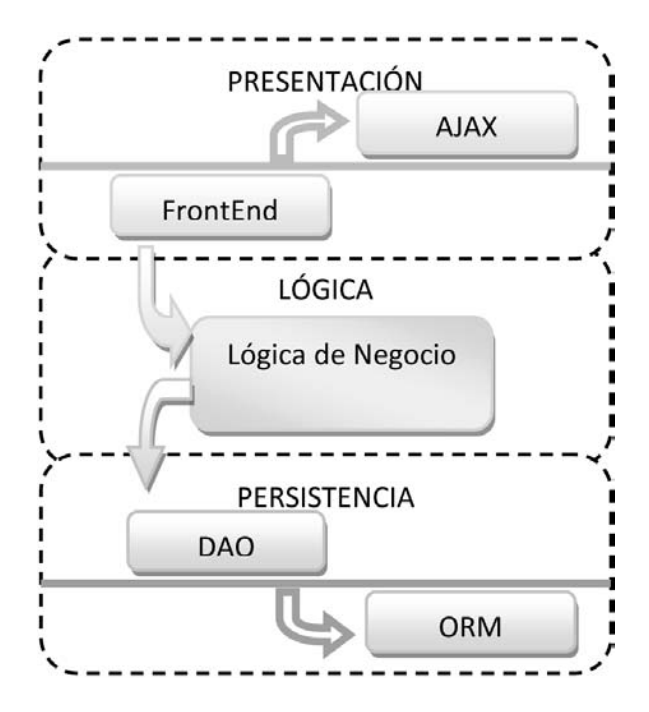


Figura 19. Arquitectura multicapa usando AJAX y ORM.

CAPÍTULO 11

Interfaz gráfica de usuario *(GUI)*

Swing es un componente de Java que se ocupa de construir interfaces gráficas de usuario. *Swing* es la evolución de *AWT* que ha estado presente en Java desde la versión 1.0. *Swing* se incorporó a Java en la versión 1.2.

En la versión 1.6 de Java, todos los desarrollos de interfaz gráfica se realizan mediante *Swing*, debido a la potencialidad que este componente ofrece. De esta forma, todos los componentes que pertenecen a *Swing* pertenecen al paquete **javax.swing**. La siguiente figura presenta la jerarquía de herencia de los componentes que posee *Swing*. Las clases en cursiva hacen referencia a clases abstractas.

En la figura 20 se puede apreciar que todas las clases del paquete *Swing* heredan de la clase *Component* que pertenece al paquete *AWT*. Esta clase posee una gran cantidad de métodos que son utilizados por todas las clases del paquete *Swing*. Estos métodos permiten proporcionar a todas las clases una gran cantidad de servicios.

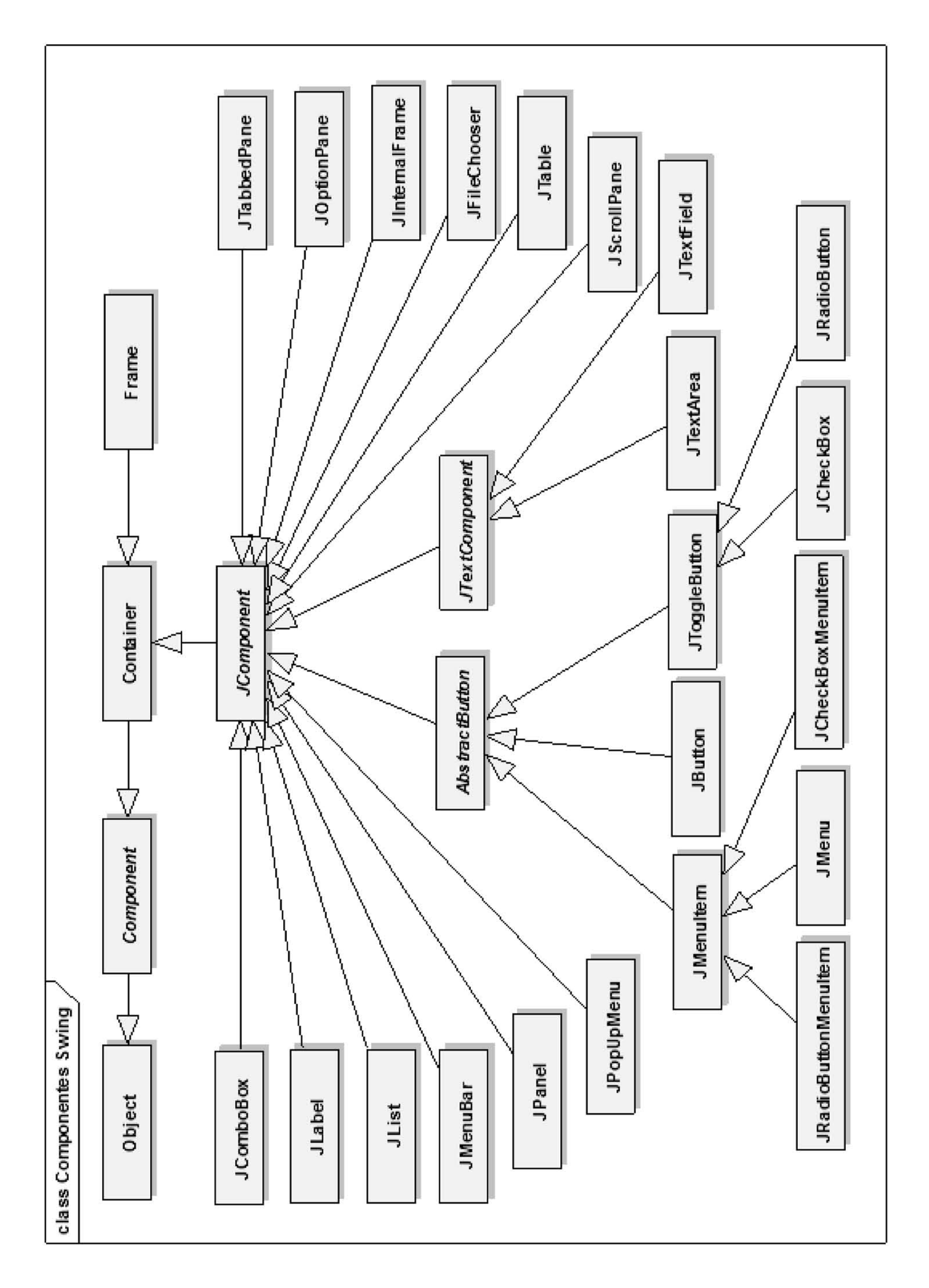


Figura 20. Jerarquía de herencia de los componentes de *Swing*

Tabla 32. Métodos principales de la clase *Component*

Retorno	Método	Descripción
void	*add(PopupMenu popup)*	Adiciona un menú emergente al componente.
void	*addComponent Listener (ComponentListener l)*	Agrega al componente un objeto para manipulación de eventos.
void	*addFocusListener (FocusListener l)*	Agrega al componente un objeto para manipulación de eventos de foco.
void	*addKeyListener (KeyListener l)*	Agrega al componente un objeto para manipulación de eventos de teclado.
void	*addMouseListener (MouseListener l)*	Agrega al componente un objeto para manipulación de eventos de los botones del *mouse*.
void	*addMouseMotion Listener (MouseMotion Listener l)*	Agrega al componente un objeto para manipulación de eventos de movimiento del *mouse*.
void	*addMouseMotionListener (MouseMotionListener l)*	Agrega al componente un objeto para manipulación de eventos de movimiento del *mouse*.
float	*getAlignmentX()*	Retorna un valor que indica la alineación en el eje X.
float	*getAlignmentY()*	Retorna un valor que indica la alineación en el eje Y.
Color	*getBackground()*	Retorna el color de fondo del componente.
Font	*getFont()*	Retorna la fuente de texto del componente.

int	*getHeight()*	Retorna el alto en píxeles del componente.
Dimension	*getMaximumSize()*	Retorna el tamaño máximo del componente.
Dimension	*getMinimumSize ()*	Retorna el tamaño mínimo del componente.
String	*getName()*	Retorna el nombre del componente.
Dimension	*getPreferredSize()*	Retorna el tamaño preferido del componente. Este hace referencia al tamaño inicial.
Dimension	*getSize()*	Retorna el tamaño actual del componente.
int	*getWidth()*	Retorna el ancho del componente.
int	*getX()*	Retorna la coordenada X de la posición del componente.
int	*getY()*	Retorna la coordenada Y de la posición del componente.
boolean	*hasFocus()*	Retorna verdadero si el componente posee el foco en tiempo de ejecución de la aplicación.
boolean	*isEnabled()*	Retorna verdadero si el componente se encuentra habilitado.
boolean	*isVisible()*	Retorna verdadero si el componente esta visible.
void	*paint(Graphics g)*	Permite pintar en el componente.
void	*remove(MenuComponent popup)*	Retira el menú emergente especificado del componente.
void	*setBackground(Color c)*	Coloca color de fondo al componente.

void	1.1.1. *setBounds(int x, int y, int width, int height)*	Coloca el componente en la posición determinada por las coordenadas x,y con el tamaño determinado por el ancho y alto.
void	*setEnabled(boolean b)*	Habilita o inhabilita el componente.
void	*setFont(Font f)*	Coloca fuente de texto al componente.
void	*setName(String name)*	Coloca nombre al componente.
void	*setSize(int width, int height)*	Coloca tamaño al componente.
void	*setVisible(boolean b)*	Muestra u oculta el componente.

11.2 Contenedores

Un contenedor es un elemento gráfico que permite agrupar diferentes elementos gráficos. Toda aplicación debe tener al menos un contenedor para que a través de este pueda iniciarse la aplicación.

11.2.1 *JFrame*

Un *JFrame* es un contenedor que se comporta como una ventana, la cual puede tener propiedades físicas. Estas propiedades pueden estar dadas por el tamaño, color y posición, entre otras.

Para implementar un *JFrame* es necesario crear una clase que herede de la clase *JFrame* del paquete **javax.swing.**

La sintaxis para crear un *JFrame* es la siguiente:

```
package interfazGrafica;

import javax.swing.JFrame;
```

```
import javax.swing.WindowConstants;

public class MiJFrame extends JFrame {

   public static void main(String[] args) {
       MiJFrame frame = new MiJFrame();
       frame.setVisible(true);
   }

   public MiJFrame() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Mi JFrame");
       setSize(400, 300);
   }
}
```

Al ejecutar la aplicación se despliega el *JFrame* con tamaño de 400 píxeles de ancho y 300 píxeles de alto y título: Mi *JFrame*.

El método *setDefaultCloseOperation* permite configurar el *JFrame* para que al dar clic en la X superior derecha, el *JFrame* se cierre. El *JFrame* se presenta como se muestra en la Figura 21.



Figura 21. JFrame

11.2.2 *JInternalFrame*

Un *JInternalFrame* es un contenedor que se comporta como una ventana interna, es decir, una ventana que se puede abrir solo dentro de un *JFrame*.

A través de un *JInternalFrame* es posible implementar aplicaciones *MDI* (*Multiple Interface Document*), debido a que es posible abrir varios *JInternalFrame* dentro de un *JFrame*, en donde cada uno de ellos provee funcionalidades independientes a la aplicación.

Para que un *JFrame* pueda contener un *JInternalFrame* es necesario tener dentro del *JFrame* otro contenedor especial denominado *JDesktopPane*. Este contenedor se adiciona al *JFrame* con la siguiente sintaxis:

```
JDesktopPane desktopPane = new JDesktopPane();
getContentPane().add(desktopPane);
```

Desde el *JFrame* debe crearse un instancia al *JInternalFrame*. Para adicionar la instancia al *JFrame* se debe hacer a través del *JDesktopPane* con la siguiente sintaxis:

```
FInterno frame = new FInterno();
desktopPane.add(frame);
```

La siguiente implementación presenta la creación de una aplicación simple basada en *JInternalFrame.*

Clase *FPrincipal*

```
package interfazGrafica.internalFrame;

import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JDesktopPane desktopPane;
```

```
    public static void main(String[] args) {
        FPrincipal frame = new FPrincipal();
        frame.setVisible(true);
    }

    public FPrincipal() {
        initGUI();
        FInterno1 frame1 = new FInterno1();
        this.desktopPane.add(frame1);
        FInterno2 frame2 = new FInterno2();
        this.desktopPane.add(frame2);
    }

    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Principal");
        {
            this.desktopPane = new JDesktopPane();
            getContentPane().add(desktopPane);
        }
        setSize(400, 300);
    }
}
```

Clase *FInterno1*

```
package interfazGrafica.internalFrame;

import javax.swing.JInternalFrame;

public class FInterno1 extends JInternalFrame {

    public FInterno1() {
        initGUI();
    }

    private void initGUI() {
        setSize(200, 100);
        setVisible(true);
        setTitle("Frame Interno 1");
    }
}
```

Clase *FInterno2*

```
package interfazGrafica.internalFrame;

import javax.swing.JInternalFrame;

public class FInterno2 extends JInternalFrame {

   public FInterno2() {
       initGUI();
   }

   private void initGUI() {
       setSize(200, 100);
       setVisible(true);
       setTitle("Frame Interno 2");
   }
}
```

Al ejecutar la aplicación el resultado es como se muestra en la Figura 22.

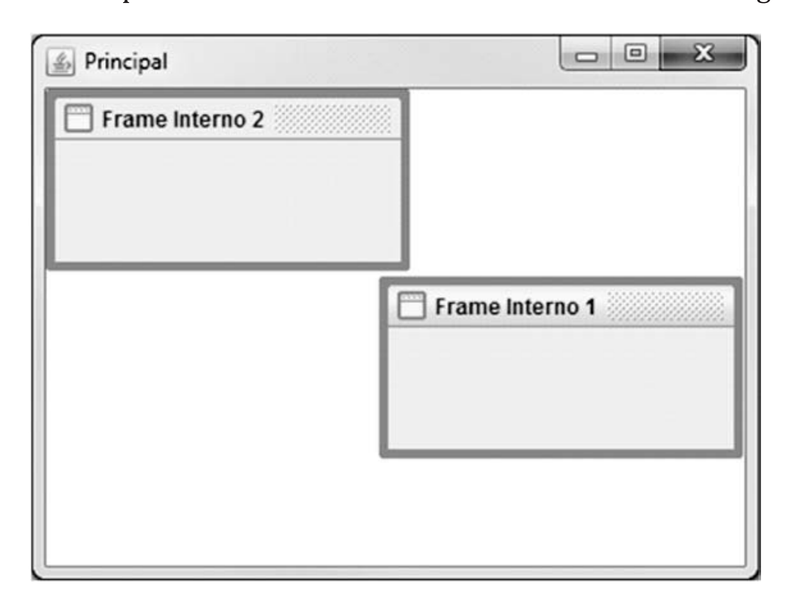


Figura 22. *JInternalFrame*

11.2.3 JPanel

Un *JPanel* es un contenedor que tiene muchas aplicaciones. Dentro de las aplicaciones más comunes están, el permitir agregar componentes para que puedan ser organizados gráficamente de una forma determinada. Otra aplicación común es utilizar el *JPanel* como pizarra para gráficos. Un panel también puede tener un título de acuerdo al uso que se le esté dando. La implementación de un *JPanel* en un *JFrame* es la siguiente:

```
package interfazGrafica.panel;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.WindowConstants;
import javax.swing.border.TitledBorder;

public class FPrincipal extends JFrame {
   private JPanel panel;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       this.setTitle("Frame Principal");
       {
          panel = new JPanel();
          panel.setBorder(new TitledBorder("Titulo del Panel"));
          getContentPane().add(panel);
       }
       setSize(400, 300);

   }
}
```

El resultado es como se muestra en la Figura 23.



Figura 23. *JPanel*

11.2.4 JTabbedPane

Un *JTabbedPane* es un contenedor de pestañas, en donde cada pestana se debe construir con un *JPanel*. La forma de agregar un panel, a un panel de pestañas, es haciendo uso del método *addTab*.

La sintaxis para crear un panel de pestañas y para crear paneles que se agreguen a las pestañas es la siguiente:

```
JTabbedPane panelPestanas = new JTabbedPane();
JPanel panel1 = new JPanel();
panelPestanas.addTab("Pestaña 1", panel1);
JPanel panel2 = new JPanel();
panelPestanas.addTab("Pestaña 2", new ImageIcon("img/
informacion.png"),panel2);
```

En el código anterior se puede apreciar que el primer panel se agregó con un método sobrecargado de dos parámetros, que contienen el texto de la pestaña y el panel correspondiente. El segundo panel se agregó con un método sobrecargado de tres parámetros que incluye adicionalmente, un ícono que se envía con la ruta donde se encuentra dicha imagen. La imagen debe encontrarse en una carpeta dentro del

proyecto. La imagen puede ser de extensiones *jpg*, *png*, *gif* o *ico*. Se recomienda utilizar imágenes con resolución de 16 x 16 píxeles.

La implementación de un JTabbedPane en un JFrame es la siguiente.

```
package interfazGrafica.tabbedPane;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JPanel panel1;
   private JPanel panel2;
   private JTabbedPane panelPestanas;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       {
          panelPestanas = new JTabbedPane();
          getContentPane().add(panelPestanas);
          {
              panel1 = new JPanel();
              panelPestanas.addTab("Pestaña 1", panel1);
          }
          {
              panel2 = new JPanel();
              panelPestanas.addTab("Pestaña 2",
              new ImageIcon("img/informacion.png"), panel2);

              }
          }
       setSize(400, 300);
   }
}
```

El resultado es como se muestra en la Figura 24.

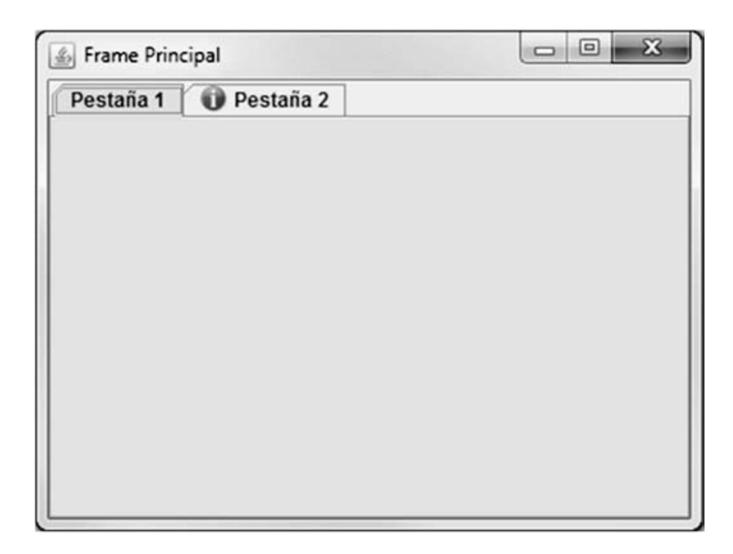


Figura 24. *JTabbedPane*

11.3 Componentes

Existen diferentes componentes que pueden ser utilizados en una aplicación para presentar la información y proveer servicios a la aplicación.

Idealmente, los componentes deben agregarse dentro de un panel que los contenga con el fin de organizar y separar los diferentes componentes, de acuerdo a sus características o servicios.

Todos los componentes pueden tener propiedades físicas. Estas propiedades pueden estar dadas por el tamaño, color y posición, entre otros.

Así mismo, el método *getContentPane* permite agregar un componente a un *JFrame*. El método *setBound* permite colocarle posición y tamaño al componente con respecto al *JFrame*, donde los dos primeros parámetros refieren a la posición izquierda y arriba, y los dos últimos parámetros refieren al tamaño ancho y alto medido en píxeles.

11.3.1 *JButton*

Un *JButton* es un botón, el cual provee un servicio fundamental de invocar un método, cuando el usuario hace clic sobre dicho botón. Un botón, generalmente, se identifica por el texto que hay sobre él. La sintaxis para crear un botón es la siguiente:

```
JButton boton = new JButton("Mi botón");
```

Si se desea cambiar el texto se usa el método *setText* de la siguiente forma:

```
boton.setText("Nuevo texto");
```

11.3.2 *JTextField*

Un *JTextField* es un cuadro de texto que provee un servicio fundamental como es permitir al usuario introducir o visualizar texto. La sintaxis para crear un cuadro de texto es la siguiente:

```
JTextFieldcuadroTexto = new JTextField("Texto");
```

Si se desea cambiar el texto se usa el método *setText* de la siguiente forma:

```
cuadroTexto.setText("Nuevo texto");
```

11.3.3 *JLabel*

Un *JLabel* es una etiqueta, la cual provee únicamente la opción de visualizar información.

La sintaxis para crear una etiqueta es la siguiente:

```
JLabel etiqueta = new JLabel("Texto");
```

Si se desea cambiar el texto se usa el método *setText* de la siguiente forma:

```
etiqueta.setText("Nuevo texto");
```

11.3.4 *JRadioButton*

Un *JRadioButton* es un componente que permite realizar una sola selección entre un conjunto de opciones.

La sintaxis para crear un radio botón es la siguiente:

```
JRadioButton radioBoton = new JRadioButton("Texto de opcion");
```

Para poder realizar la activación correcta del radio botón, en donde solo debe haber un botón activo a la vez, se debe hacer una agrupación mediante un *ButtonGroup*. Para agrupar tres radio botones, la sintaxis es la siguiente:

```
JRadioButton radioBoton1 = new JRadioButton("Opcion 1");
JRadioButton radioBoton2 = new JRadioButton("Opcion 2");
JRadioButton radioBoton3 = new JRadioButton("Opcion 3");
ButtonGroup grupoRadioBoton = new ButtonGroup();
grupoRadioBoton.add(radioBoton1);
grupoRadioBoton.add(radioBoton2);
grupoRadioBoton.add(radioBoton3);
```

El *JRadioButton* posee dos métodos para la manipulación de la selección del componente que son:

- *isSelected()*. Retorna verdadero si el *JRadioButton* se encuentra seleccionado y falso si no se encuentra seleccionado.
- *setSelected(boolean b)*. Selecciona el *JRadioButton* si el parámetro es verdadero. Al usar este método se retira de forma automática la selección de los demás *JRadioButton* que hagan parte del *ButtonGroup*.

11.3.5 *JCheckBox*

Un *JCheckBox* es un componente que permite realizar múltiples selecciones de opciones.

La sintaxis para crear un *checkbox* es la siguiente:

```
JCheckBox checkBox = new JCheckBox("Texto de opcion");
```

El *JCheckBox* posee dos métodos para la manipulación de la selección del componente que son:

- *isSelected()*. Retorna verdadero si el *JCheckBox* se encuentra seleccionado y falso si no se encuentra seleccionado.
- *setSelected(boolean b)*. Selecciona el *JCheckBox* si el parámetro es verdadero.

11.3.6 *JTextArea*

Un *JTextArea* es un área de texto que permite tener múltiples líneas de texto.

La sintaxis para crear un área de texto es la siguiente:

```
JTextAreaareaTexto = new JTextArea();
```

Si se desea asignar el número de líneas de texto para el área de texto, se debe hacer uso del método *setRows*, el cual recibe un parámetro entero que indica el número de filas que se desean asignar al área de texto.

```
areaTexto.setRows(10);
```

11.3.7 *JList*

Un *JList* es una lista, la cual permite visualizar un conjunto de textos. Estos textos pueden ser seleccionados en tiempo de ejecución de forma simple o múltiple, es decir, el usuario puede seleccionar uno o varios textos. La sintaxis para crear una lista es la siguiente:

```
JListlista = new JList();
```

Para agregar información en una lista, se pueden utilizar varias técnicas. Las técnicas más utilizadas son las siguientes:

- Crear un arreglo de *String* y enviarlo al *JList* como parámetro en el constructor. La implementación de esta solución es la siguiente.

```
String[] textos = {"Texto 1","Texto 2","Texto 3","..","Texto n"};
JList lista = new JList(textos);
```

- Crear un *DefaultListModel*, y a través del método *addElement*, agregar los textos que se requieran adicionar a la lista. El *DefaultListModel* se asigna a la lista a través del método *setModel* de la clase *JList*. La implementación de esta solución es la siguiente:

```
JList lista = new JList();
DefaultListModel modeloLista = new  DefaultListModel();
modeloLista.addElement("Texto 1");
modeloLista.addElement("Texto 2");
modeloLista.addElement("Texto 3");
modeloLista.addElement("..");
modeloLista.addElement("Texto n");
lista.setModel(modeloLista);
```

- Usar la interfaz *ListModel* y la clase *DefaultComboBoxModel*, creando un objeto referencia de *ListModel* e instancia de *DefaultComboBoxModel*, asignando en su constructor un arreglo de *String* con los textos requeridos. El *ListModel* se asigna a la lista a través del método *setModel* de la clase *JList*. La implementación de esta solución es la siguiente:

```
ListModel modeloLista = new DefaultComboBoxModel(new String[]
{"Texto 1","Texto 2","Texto 3","..","Texto n"});
JList lista = new JList();
lista.setModel(modeloLista);
```

La implementación de una aplicación de las tres soluciones anteriores es la siguiente:

```
package interfazGrafica.list;

import javax.swing.DefaultComboBoxModel;
import javax.swing.DefaultListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.ListModel;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JList lista1;
   private JList lista2;
   private JList lista3;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       getContentPane().setLayout(null);
       {
          String[] textos = {"Lista 1", "Texto 1", "Texto 2",
          "Texto 3", "..", "Texto n"};
          lista1 = new JList(textos);
          getContentPane().add(lista1);
          lista1.setBounds(40, 30, 80, 150);
       }
       {
          lista2 = new JList();
          DefaultListModel modeloLista = new  DefaultListModel();
          modeloLista.addElement("Lista 2");
          modeloLista.addElement("Texto 1");
          modeloLista.addElement("Texto 2");
          modeloLista.addElement("Texto 3");
          modeloLista.addElement("..");
          modeloLista.addElement("Texto n");
          lista2.setModel(modeloLista);
          getContentPane().add(lista2);
          lista2.setBounds(160, 30, 80, 150);
       }
       {
```

```
        ListModel modeloLista = new DefaultComboBoxModel(
            new String[] {"Lista 3", "Texto 1", "Texto 2", "Texto 3",
            "..", "Texto n"});
        lista3 = new JList();
        lista3.setModel(modeloLista);
        getContentPane().add(lista3);
        lista3.setBounds(280, 30, 80, 150);
    }
    setSize(400, 300);
  }
}
```

El resultado es como se presenta en la Figura 25.

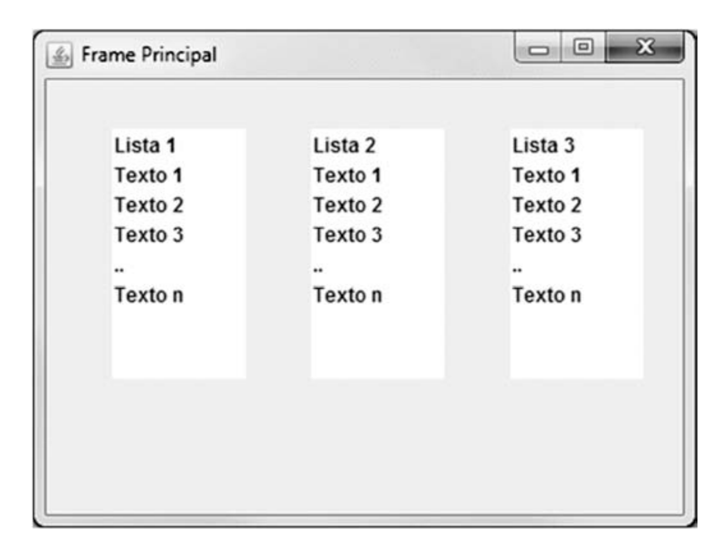


Figura 25. JList

El *JList* posee cuatro métodos para la manipulación de la selección del componente que son:

- *getSelectedIndex()*. Retorna el índice del elemento seleccionado.
- *getSelectedIndices()*. Retorna un arreglo de enteros con los índices de los elementos seleccionados.
- *getSelectedValue()*. Retorna un objeto con el valor del elemento seleccionado.

- *getSelectedValues()*. Retorna un arreglo de objetos con los valores de los elementos seleccionados.

11.3.8 JComboBox

Un *JComboBox* es un componente que combina un cuadro de texto con una lista. Se usa con frecuencia en situaciones donde se requiere seleccionar y visualizar solo un resultado de la lista.

La sintaxis para crear un combo es la siguiente:

```
JComboBox combo = new JComboBox();
```

Para agregar información en un combo se pueden utilizar varias técnicas. La técnica más utilizadas consisten en usar la interfaz *ComboBoxModel* y la clase *DefaultComboBoxModel,* creando un objeto referencia de *ComboBoxModel* e instancia de *DefaultComboBoxModel,* asignando en su constructor un arreglo de *String* con los textos requeridos. El *ComboBoxModel* se asigna al combo a través del método *setModel* de la clase *JComboBox*. La implementación de esta solución es la siguiente:

```
package interfazGrafica.comboBox;

import javax.swing.ComboBoxModel;
import javax.swing.DefaultComboBoxModel;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JComboBox combo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }
```

```
    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Frame Principal");
        getContentPane().setLayout(null);
        {
           ComboBoxModel comboModel =
               new DefaultComboBoxModel(
                  new String[] {"Seleccione", "Texto 1",
                  "Texto 2", "Texto 3", "..", "Texto n"});
           combo = new JComboBox();
           getContentPane().add(combo);
           combo.setModel(comboModel);
           combo.setBounds(120, 50, 110, 20);
        }
        setSize(400, 300);
    }
}
```

El resultado es como se presenta en la Figura 26.

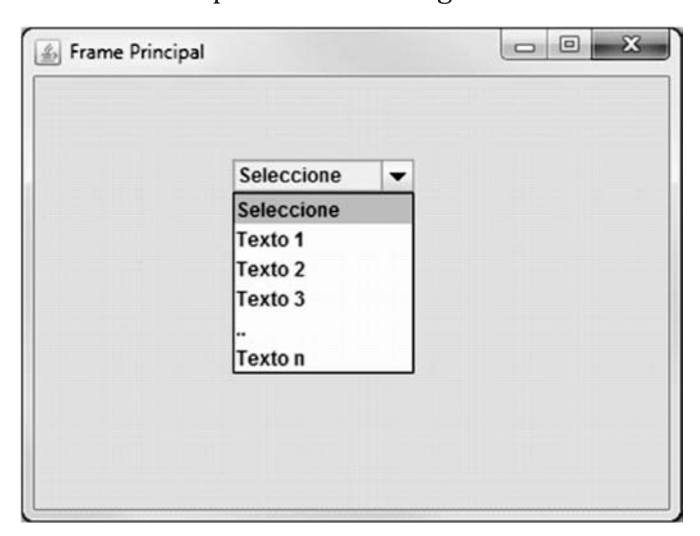


Figura 26. JComboBox

El *JList* posee dos métodos para la manipulación de la selección del componente que son:

1. *getSelectedIndex()*. Retorna el índice del elemento seleccionado.

2. *getSelectedItem()*. Retorna el objeto del elemento seleccionado.

11.3.9 JTable

Un *JTable* es un componente que permite visualizar información en forma de tabla. La tabla puede contener varias filas y columnas. En un *JTable* es posible seleccionar filas, ordenar filas a partir de una columna seleccionada y muchas otras funciones.

La sintaxis para crear una tabla es la siguiente:

```
JTable tabla = new JTable();
```

Para asignar información a una tabla es necesario hacer uso de la interfaz *TableModel* y la clase *DefaultTableModel,* creando un objeto referencia de *TableModel* e instancia de *DefaultTableModel,* asignando en su constructor una matriz tipo *String* que contenga los datos y un arreglo tipo *String* con los títulos. Este modelo se debe asignar a la tabla mediante el método *setModel.*

La sintaxis para asignar un *TableModel* a un *JTable* es la siguiente:

```
String [][] datos = {{"Dato Fila 0 Columna 0","Dato Fila 0 
Columna 1","Dato Fila 0 Columna 2"}, {"Dato Fila 1 Columna 
0","Dato Fila 1 Columna 1","Dato Fila 1 Columna 2"}};
String [] titulos = new String[] { "Columna 0", "Columna 2", 
"Columna 3"};
TableModel modeloTabla = new DefaultTableModel(datos, titulos);
tabla.setModel(modeloTabla);
```

La manera más usual de utilizar una tabla es a través del contenedor *JScrollPane*, el cual permite que se activen de forma automática las barras de desplazamiento para visualizar toda la información de la tabla, en el caso que la información no pueda ser visualizada por el tamaño de la tabla. Para asignar una tabla a un *JScrollPane*, se hace uso del método *setViewportView.* Al utilizar un *JScrollPane* es necesario asignar el tamaño interno de la tabla mediante el método

setPreferredSize, el cual recibe como parámetro un objeto instancia de la clase *Dimensión* que recibe como parámetros, el ancho y alto de la tabla.

La sintaxis para asignar un *JTable* a un *JScrollPane* es la siguiente:

```
JScrollPane scrollPane = new JScrollPane();
scrollPane.setViewportView(tabla);
tabla.setPreferredSize(new Dimension(350,datos.length*16));
```

En el ejemplo anterior, el parámetro *datos.length*16*, permite configurar el alto de la tabla de acuerdo al número de filas de la misma multiplicado por 16 píxeles que equivale a la altura de una fila en el tipo y tamaño de fuente por defecto.

Adicionalmente, es posible ordenar las filas de la tabla mediante la clase *TableRowSorter*. Es necesario crear una instancia de *TableRowSorter* enviando como parámetro al constructor el modelo de la tabla. Posteriormente, se le asigna a la tabla mediante el método *setRowSorter.*

La sintaxis para asignar un *TableRowSorter* a un *JTable* es la siguiente:

```
TableRowSorter ordenador=new TableRowSorter(modeloTabla);
tabla.setRowSorter(ordenador);
```

Un ejemplo de implementación de una tabla en un *frame*, con todas las características presentadas es el siguiente:

```
package interfazGrafica.table;

import java.awt.Dimension;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.WindowConstants;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;
```

```
public class FPrincipal extends JFrame {
   private JTable tabla;
   private JScrollPane scrollPane;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       getContentPane().setLayout(null);
       {
          scrollPane = new JScrollPane();
          getContentPane().add(scrollPane);
          scrollPane.setBounds(40, 20, 320, 120);
          {
              String [][] datos = {{"10","Hector",
              "Florez","123"}, {"20","Arturo","Fernandez","456"},
              {"30","Juan","Valdez","789"}, {"40","Pepito",
              "Perez","987"},{"50","Pedro","Picapiedra","123"},
              {"60","Pablo","Marmol","456"}, {"70","Homero",
              "Simpson","789"}, {"80","Bart","Simpson","987"}};
              String [] titulos = new String[] { "Identificacion",
              "Nombre", "Apellido", "Telefono"};
              TableModel modeloTabla = new DefaultTableModel(datos, titulos);
              tabla = new JTable();
              tabla.setModel(modeloTabla);
              tabla.setPreferredSize(new Dimension(350,datos.length*16));
              scrollPane.setViewportView(tabla);
              TableRowSorter ordenador = new TableRowSorter(modeloTabla);
              tabla.setRowSorter(ordenador);
          }
       }
       setSize(400, 300);
   }
}
```

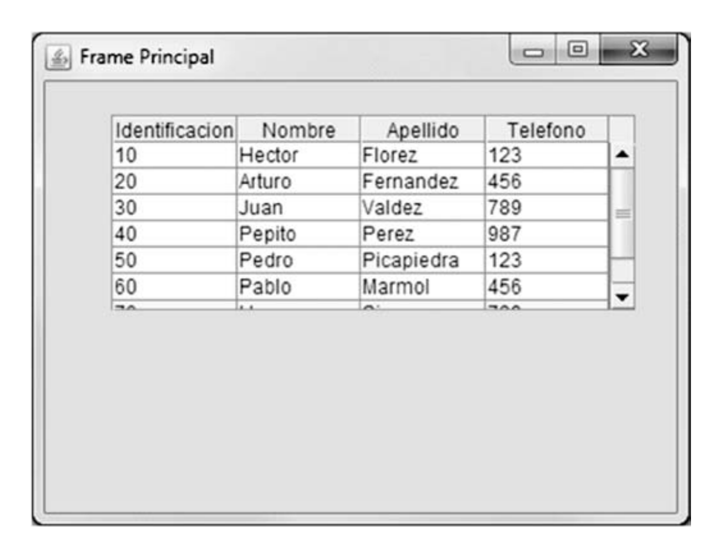
El resultado es como se presenta en la Figura 27.

Frame Principal

Identificacion	Nombre	Apellido	Telefono
10	Hector	Florez	123
20	Arturo	Fernandez	456
30	Juan	Valdez	789
40	Pepito	Perez	987

Figura 27. *JTable*

Al agregar más información en la tabla se activan de forma automática, las barras de desplazamiento necesarias como se muestra en la Figura 28.



Frame Principal

Identificacion	Nombre	Apellido	Telefono
10	Hector	Florez	123
20	Arturo	Fernandez	456
30	Juan	Valdez	789
40	Pepito	Perez	987
50	Pedro	Picapiedra	123
60	Pablo	Marmol	456

Figura 28. *JTable* con *JScrollPane*

Al hacer clic sobre el título de la columna “*Nombre*” se realiza el ordenamiento automático de los datos de la tabla, como se muestra en la Figura 29.

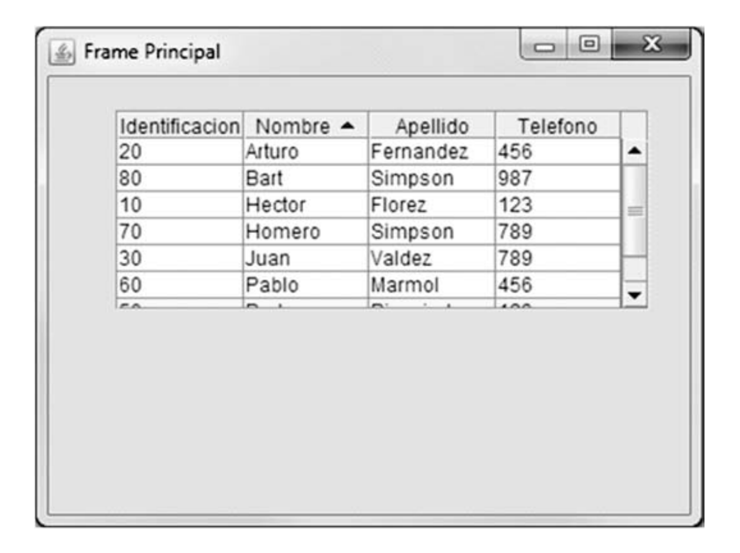


Figura 29. JTable ordenado

11.4 Cuadros de diálogo

11.4.1 *JOptionPane*

La clase *JOptionPane* contiene una gran cantidad de atributos y métodos estáticos que permiten generar diferentes tipos de cuadros de diálogo. Reciben diferentes parámetros de acuerdo al tipo de cuadro de mensaje, sin embargo, todos los cuadros de mensaje reciben, en su primer parámetro, un componente que hace referencia al *JFrame* del cual depende el cuadro de diálogo. Generalmente, el cuadro de diálogo depende del *JFrame* que hace uso de dicho cuadro, por tal razón, el primer parámetro puede contener la sentencia *this*. Estos cuadros de diálogo se clasifican en los siguientes tipos:

- **Cuadros de diálogo de mensaje.** Un cuadro de mensaje presenta una información al usuario como resultado de una

operación. Este mensaje está acompañado de un ícono que permite indicar, si el mensaje es de información, error o advertencia.

- **Cuadros de diálogo de confirmación.** Un cuadro de confirmación provee un mensaje más tres botones que son **SI**, **NO** y **CANCELAR.** Cada uno de estos botones poseen un valor que puede ser capturado en la aplicación.

- **Cuadros de diálogo de entrada de información**. Un cuadro de entrada provee un cuadro de texto para que el usuario digite allí una información, que va a ser capturada en una cadena de caracteres en la aplicación.

- **Cuadros de diálogo de opciones**. Un cuadro de opciones provee un conjunto de botones que se envían a través de un arreglo. Este cuadro puede tener un ícono personalizado, mensaje y título. El cuadro retorna el índice de la opción seleccionado por el usuario a través de un clic sobre un botón.

Cuadro de mensaje de información

```
JOptionPane.showMessageDialog(this, "Mensaje de informacion del
cuadro de dialogo", "Titulo", JOptionPane.INFORMATION_MESSAGE);
```

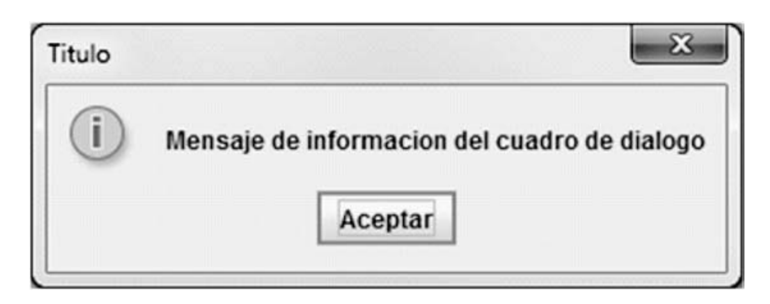


Figura 30. *JOptionPane*. Cuadro de mensaje de información

Cuadro de mensaje de error

```
JOptionPane.showMessageDialog(this, "Mensaje de error del cuadro
de dialogo", "Titulo", JOptionPane.ERROR_MESSAGE);
```



Figura 31. *JOptionPane*. Cuadro de mensaje de error

Cuadro de mensaje de advertencia

```
JOptionPane.showMessageDialog(this, "Mensaje de advertencia del
cuadro de dialogo", "Titulo", JOptionPane.WARNING_MESSAGE);
```



Figura 32. *JOptionPane*. Cuadro de mensaje de advertencia

Cuadro de mensaje de confirmación

```
int accion=JOptionPane.showConfirmDialog(this, "Mensaje de
confirmacion");
if (accion==JOptionPane.YES_OPTION){
   //Entra si hace clic en SI
}else if (accion==JOptionPane.NO_OPTION){
   //Entra si hace clic en NO
```

```
}else if (accion==JOptionPane.CANCEL_OPTION){
   //Entra si hace clic en CANCELAR
}
```

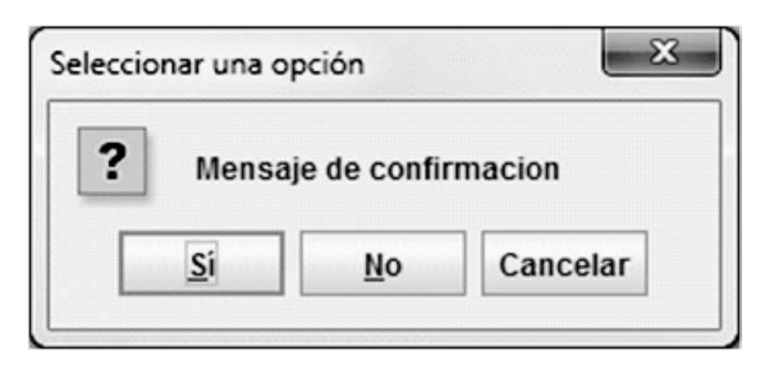


Figura 33. *JOptionPane*. Cuadro de mensaje de confirmación

Cuadro de mensaje de entrada de información

```
String informacion = JOptionPane.showInputDialog(this, "Mensaje
de solicitud de informacion");
```

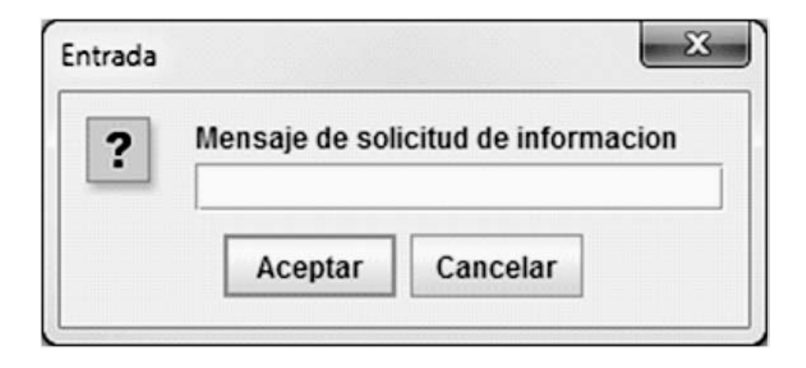


Figura 34. *JOptionPane*. Cuadro de mensaje de entrada de información

Cuadro de mensaje de opción con botones

```
String []opciones={"Opcion 1","Opcion 2","Opcion 3","Opcion n"};
ImageIcon imagen = new ImageIcon("img/opcion.png");
int indice = JOptionPane.showOptionDialog(this, "Mensaje de
seleccion de opciones", "Titulo", 1, 1, imagen, opciones, "1");
```



Figura 35. *JOptionPane*. Cuadro de mensaje de opción con botones

Cuadro de mensaje de opción con comboBox

```
String []opciones={"Opcion 1","Opcion 2","Opcion 3","Opcion n"};
JOptionPane.showInputDialog(this, "Mensaje de seleccion de
opciones", "Titulo", JOptionPane.PLAIN_MESSAGE, null, opciones,
"?");
```



Figura 36. *JOptionPane*. Cuadro de mensaje de opción con comboBox

11.4.2 JFileChooser

El *JFileChooser* permite abrir diferentes tipos de cuadros de diálogo, para abrir archivos de cualquier tipo. Esta clase tiene atributos y métodos para abrir cuadros de diálogo, para guardar y abrir archivos. Los métodos *showSaveDialog* y *showOpenDialog* reciben por parámetro un contenedor. Este contenedor puede ser el *JFrame* que contiene la aplicación o un *JPanel*. Si se desea incluir el *JFrame* y el código para abrir el *JFileChooser* está en dicho *JFrame*, se debe enviar por parámetro la sentencia *this*, la cual hace referencia a dicho *JFrame*. Al colocar un contenedor, el cuadro de diálogo aparece en el centro de dicho contenedor.

Cuadro de diálogo de guardar

```
JFileChooser ventana = new JFileChooser();
int seleccion = ventana.showSaveDialog(this);
if (seleccion==JFileChooser.APPROVE_OPTION){
       File file = ventana.getSelectedFile();
}else if(seleccion == JFileChooser.CANCEL_OPTION){
       //TODO Código de cancelar
}
```

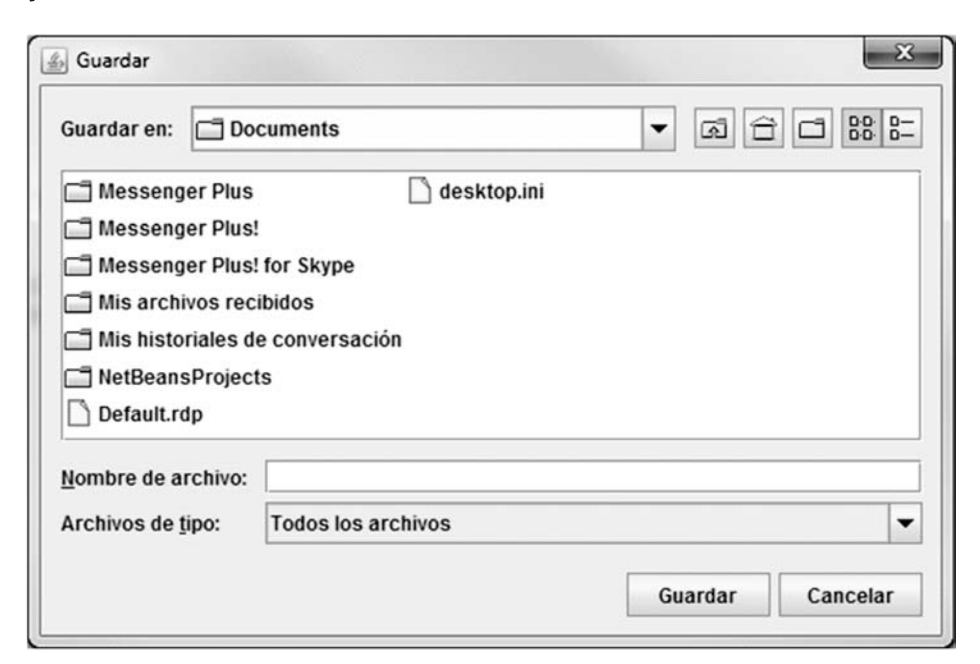


Figura 37. *JFileChooser*. Cuadro de diálogo para guardar archivo

Cuadro de diálogo de abrir

```
JFileChooser ventana = new JFileChooser();
int seleccion = ventana.showOpenDialog(this);
if (seleccion==JFileChooser.APPROVE_OPTION){
   File file = ventana.getSelectedFile();
}else if(seleccion == JFileChooser.CANCEL_OPTION){
   //TODO Código de cancelar
}
```



Figura 38. *JFileChooser*. Cuadro de diálogo para abrir archivo

Si se desea abrir el cuadro de diálogo en una ruta específica basta con colocar, en el constructor del *JFileChooser*, la ruta deseada. La implementación es la siguiente:

```
JFileChooser ventana = new JFileChooser("./img");
int seleccion = ventana.showOpenDialog(null);
if (seleccion==JFileChooser.APPROVE_OPTION){
   File file = ventana.getSelectedFile();
}else if(seleccion == JFileChooser.CANCEL_OPTION){
   //TODO Código de cancelar
}
```



Figura 39. *JFileChooser*. Cuadro de diálogo para abrir archivo con ruta relativa

El *JFileChooser* permite la inclusión de filtros en los archivos. En los casos anteriores, en el cuadro que hace referencia al tipo de archivo, solo existe la opción "*Todos los archivos*". A través de la clase abstracta *FileFilter* es posible configurar un filtro o un conjunto de filtros, de tal manera que al seleccionar uno de ellos se presenta en el cuadro de exploración solamente las carpetas y archivos que contengan la extensión del filtro seleccionado.

La forma más adecuada de implementar un filtro es creando una clase que extienda de *FileFilter*. Este procedimiento obliga a implementar los métodos abstractos *accept* y *getDescription*. La implementación es la siguiente:

```
package interfazGrafica.fileChooser;

import java.io.File;

import javax.swing.filechooser.FileFilter;

public class Filtro extends FileFilter{
```

```
    private String ext;
    private String description;

    public Filtro(){
        this.ext = ".haff";
        this.description = "Archivos Hector Florez (*.haff)";
    }

    @Override
    public String getDescription() {
        return this.description;
    }

    @Override
    public boolean accept(File f) {
        return f.getName().toLowerCase().endsWith (
        this.ext)||f.isDirectory();
    }
}
```

El método *accept* recibe como parámetro un *File* que hace referencia al archivo seleccionado, en caso que se acepte y corresponda con la extensión configurada. En caso de no coincidir, el cuadro de diálogo no se cierra.

Para asignar el filtro al *JFileChooser* es necesario utilizar el método *addChoosableFileFilter* del *JFileChooser,* enviándole como parámetro el filtro. La sintaxis es la siguiente:

```
Filtro filtro = new Filtro();
ventana.addChoosableFileFilter(filtro);
```

La implementación para abrir un cuadro de diálogo con filtro es la siguiente:

```
JFileChooser ventana = new JFileChooser();
Filtro filtro = new Filtro();
ventana.addChoosableFileFilter(filtro);
int seleccion = ventana.showOpenDialog(null);
if (seleccion==JFileChooser.APPROVE_OPTION){
    File file = ventana.getSelectedFile();
}else if(seleccion == JFileChooser.CANCEL_OPTION){
    //TODO Código de cancelar
}
```

El resultado de la asignación del filtro es como se muestra en la Figura 40.

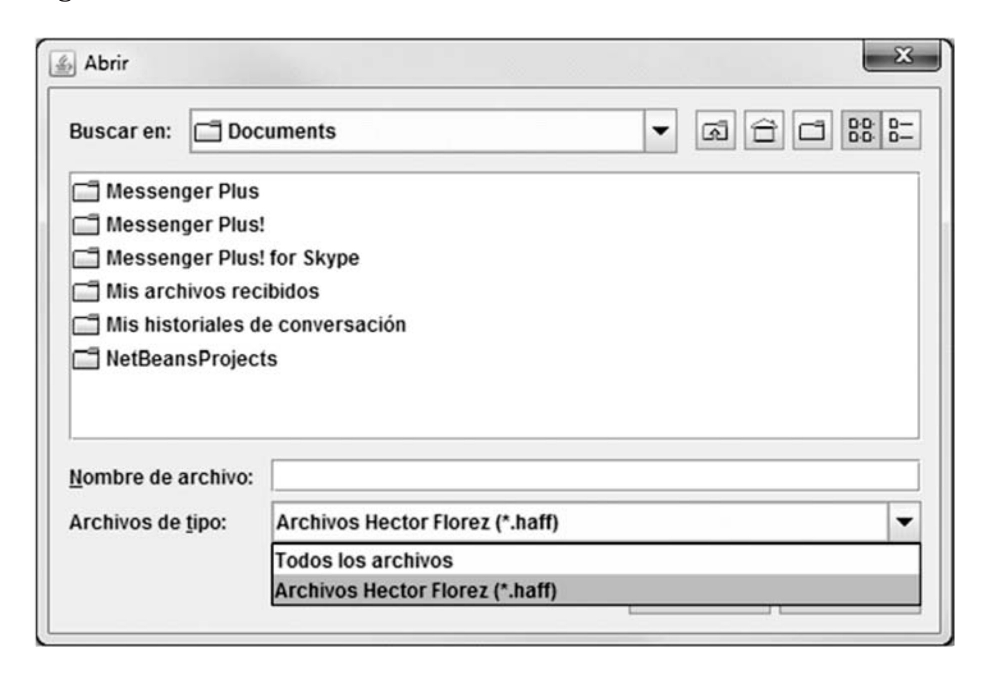


Figura 40. JFileChooser utilizando FileFilter

En la Figura 40 se puede apreciar que al desplegar el combo de tipo de archivo, se encuentra el filtro configurado y una opción por defecto.

Si se requieren múltiples filtros es posible agregar varios objetos *FileFilter*. Una estrategia es modificar el constructor de la clase filtro anteriormente creada, para que se pueda enviar la extensión y descripción del filtro en el momento de hacer la instancia. De esta forma, no se trabajaría un solo filtro por defecto. La implementación es la siguiente:

```
package interfazGrafica.fileChooser;

import java.io.File;

import javax.swing.filechooser.FileFilter;
```

```
public class Filtro extends FileFilter{
   private String ext;
   private String description;

   public Filtro(String ext, String description){
       this.ext = ext;
       this.description = description;
   }

   @Override
   public String getDescription() {
       return this.description;
   }

   @Override
   public boolean accept(File f) {
       return f.getName().toLowerCase().endsWith (
       this.ext)||f.isDirectory();
   }
}
```

Para asignar los diferentes filtros se puede crear un arreglo de filtros. Al instanciar cada objeto del arreglo se envían como parámetros la extensión y la descripción. La sintaxis es la siguiente:

```
Filtro [] filtro = new Filtro[5];
filtro[0]=new Filtro(".docx","Archivos de word 2007 (*.docx)");
filtro[1]=new Filtro(".xlsx","Archivos de excel 2007 (*.xlsx)");
filtro[2]=new Filtro(".pptx","Archivos de power point 2007
(*.pptx)");
filtro[3]=new Filtro(".pdf","Archivos pdf (*.pdf)");
filtro[4]=new Filtro(".txt","Archivos block de notas (*.txt)");
for(int i=0; i<5; i++){
   ventana.addChoosableFileFilter(filtro[i]);
}
```

La implementación para abrir un cuadro de diálogo con múltiples filtros es la siguiente:

```
JFileChooser ventana = new JFileChooser();
Filtro [] filtro = new Filtro[5];
filtro[0]=new Filtro(".docx","Archivos de word 2007 (*.docx)");
filtro[1]=new Filtro(".xlsx","Archivos de excel 2007 (*.xlsx)");
filtro[2]=new Filtro(".pptx","Archivos de power point 2007
(*.pptx)");
filtro[3]=new Filtro(".pdf","Archivos pdf (*.pdf)");
filtro[4]=new Filtro(".txt","Archivos block de notas (*.txt)");
for(int i=0; i<5; i++){
```

```
    ventana.addChoosableFileFilter(filtro[i]);
}
int seleccion = ventana.showOpenDialog(null);
if (seleccion==JFileChooser.APPROVE_OPTION){
    File file = ventana.getSelectedFile();
}else if(seleccion == JFileChooser.CANCEL_OPTION){
    //TODO Código de cancelar
}
```

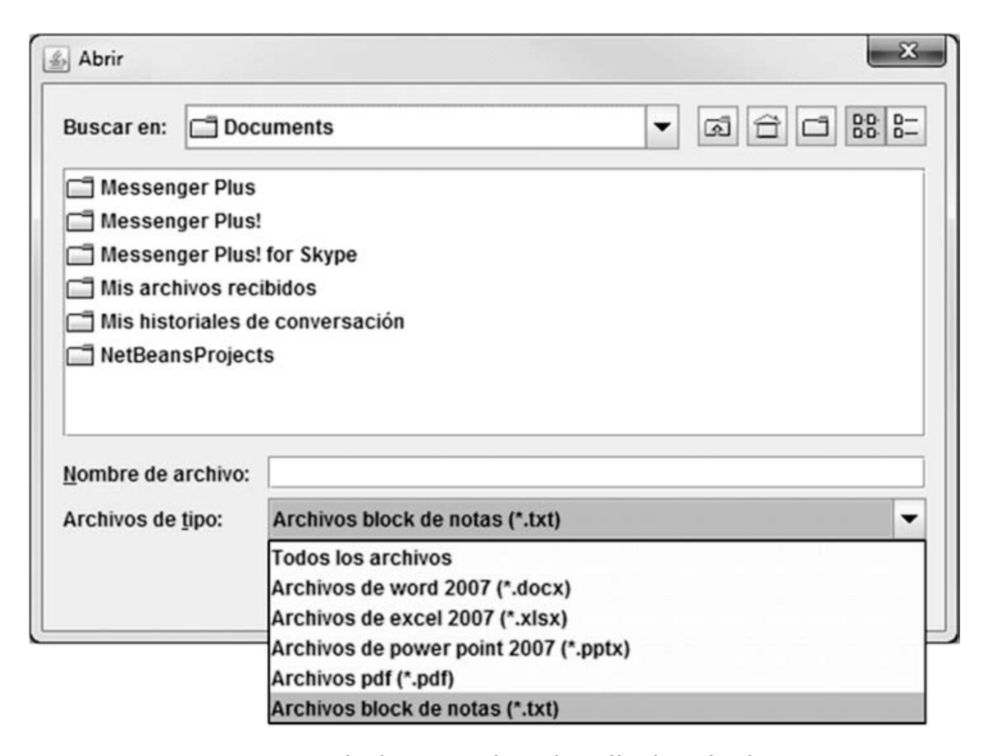


Figura 41. JFileChooser utilizando múltiples FileFilter

11.5 *Layout*

Un *layout* es un objeto que permite controlar la posición y tamaño de un conjunto de componentes en un contenedor.

11.5.1 *AbsoluteLayout*

La clase *AbsolutLayout* permite organizar elementos de forma estática en un contenedor. El elemento que se agrega al contenedor debe

establecer la posición y tamaño a través de los siguientes métodos:

- *setBounds* que permite asignar posición a través de coordenadas X, Y y tamaño, a través de atributos que indican el ancho y alto.
- *setSize* que permite asignar tamaño a través de atributos que indican el ancho y alto.
- *setPosition* que permite asignar posición a través de coordenadas las X, Y.

La sintaxis para asignar un *AbsoluteLayout* a un contenedor, como por ejemplo un panel, es la siguiente:

```
JPanel panel = new JPanel();
panel.setLayout(null);
```

La siguiente implementación permite establecer un *JFrame* con *AbsoluteLayout* en el cual se adicionan seis botones con diferentes posiciones y tamaños.

```
package interfazGrafica.layout.absolutLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JButton boton1;
   private JButton boton2;
   private JButton boton3;
   private JButton boton4;
   private JButton boton5;
   private JButton boton6;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }
```

```
    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Frame Principal");
        getContentPane().setLayout(null);
        {
            boton1 = new JButton();
            getContentPane().add(boton1);
            boton1.setText("boton 1");
            boton1.setBounds(30, 5, 101, 21);
        }
        {
            boton2 = new JButton();
            getContentPane().add(boton2);
            boton2.setText("boton 2");
            boton2.setBounds(35, 90, 97, 21);
        }
        {
            boton3 = new JButton();
            getContentPane().add(boton3);
            boton3.setText("boton 3");
            boton3.setBounds(142, 52, 110, 38);
        }
        {
            boton4 = new JButton();
            getContentPane().add(boton4);
            boton4.setText("boton 4");
            boton4.setBounds(65, 167, 123, 62);
        }
        {
            boton5 = new JButton();
            getContentPane().add(boton5);
            boton5.setText("boton 5");
            boton5.setBounds(218, 119, 117, 46);
        }
        {
            boton6 = new JButton();
            getContentPane().add(boton6);
            boton6.setText("boton 6");
            boton6.setBounds(278, 73, 93, 35);
        }
        setSize(400, 300);
    }
}
```

Al ejecutar la implementación anterior se presenta el resultado de la Figura 42.

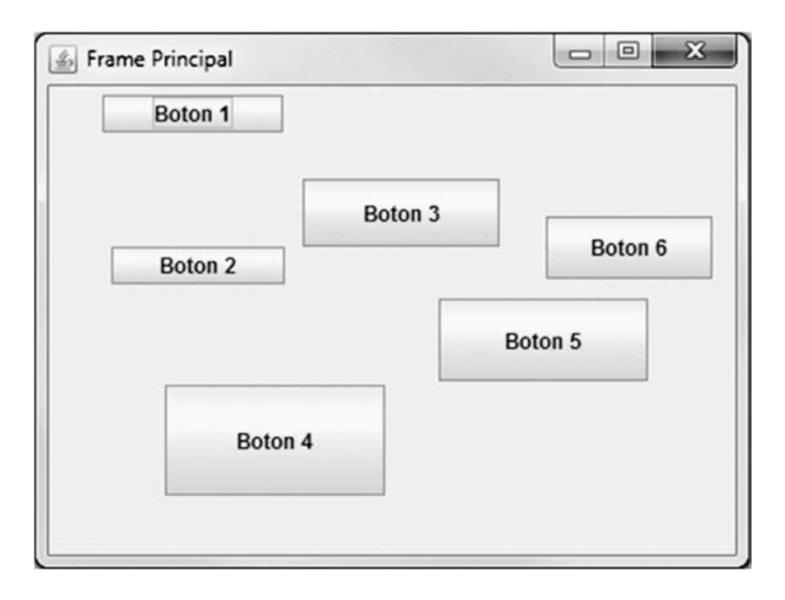


Figura 42. Absolute Layout

11.5.2 *BorderLayout*

La clase *BorderLayout* permite organizar elementos dinámicamente en un contenedor, con base en cinco posiciones. Entonces, solo se puede visualizar máximo cinco elementos en un contenedor con *BorderLayout*. Las posiciones son las siguientes:

1. *CENTER*. Esta posición permite colocar el elemento en el centro del contendor.

2. *NORTH*. Esta posición permite colocar el elemento en la parte superior del contendor.

3. *SOUTH*. Esta posición permite colocar el elemento en la parte inferior del contendor.

4. *WEST*. Esta posición permite colocar el elemento en la parte izquierda del contendor.

5. *EAST*. Esta posición permite colocar el elemento en la parte derecha del contendor.

Si un único elemento está en el contenedor con *BorderLayout,* independientemente de su posición, el elemento ocupará todo el contenedor.

La sintaxis para asignar un *BorderLayout* a un contenedor, como por ejemplo un panel, es la siguiente:

```
JPanel panel = new JPanel();
BorderLayout layout = new BorderLayout();
panel.setLayout(layout);
```

La sintaxis para agregar un elemento, como por ejemplo, un botón a un panel con *BorderLayout* en la posición *center*, es la siguiente:

```
JButton boton = new JButton("Boton");
panel.add(Boton1, BorderLayout.CENTER);
```

La siguiente implementación permite establecer un *JFrame* con *BorderLayout* en el cual se adicionan cinco botones en todas las posiciones posibles.

```
package interfazGrafica.layout.borderLayout;

import java.awt.BorderLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JButton boton1;
   private JButton boton2;
   private JButton boton3;
   private JButton boton4;
   private JButton boton5;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }
```

```
    public FPrincipal() {
        initGUI();
    }

    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Frame Principal");
        BorderLayout thisLayout = new BorderLayout();
        getContentPane().setLayout(thisLayout);
        {
            boton1 = new JButton();
            getContentPane().add(boton1, BorderLayout.CENTER);
            boton1.setText("Boton 1");
        }
        {
            boton2 = new JButton();
            getContentPane().add(boton2, BorderLayout.NORTH);
            boton2.setText("Boton 2");
        }
        {
            boton3 = new JButton();
            getContentPane().add(boton3, BorderLayout.WEST);
            boton3.setText("Boton 3");
        }
        {
            boton4 = new JButton();
            getContentPane().add(boton4, BorderLayout.EAST);
            boton4.setText("Boton 4");
        }
        {
            boton5 = new JButton();
            getContentPane().add(boton5, BorderLayout.SOUTH);
            boton5.setText("Boton 5");
        }
        pack();
        setSize(400, 300);
    }
}
```

Al ejecutar la implementación anterior se presenta el resultado de la Figura 43.

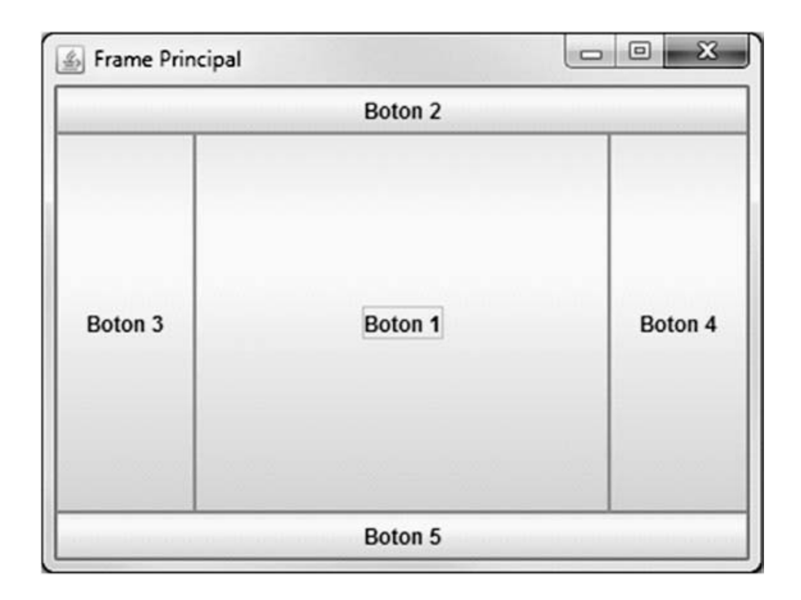


Figura 43. Border Layout

11.5.3 *FlowLayout*

La clase *FlowLayout* permite organizar elementos dinámicamente en un contenedor, en donde la posición de los elementos depende del orden en que son agregados al contenedor. Estos elementos están colocados uno seguido del otro.

La sintaxis para asignar un *FlowLayout* a un contenedor, como por ejemplo un panel, es la siguiente:

```
JPanel panel = new JPanel();
FlowLayout layout = new FlowLayout();
panel.setLayout(layout);
```

La siguiente implementación permite establecer un *JFrame* con *FlowLayout* en el cual se adicionan cinco botones.

```
package interfazGrafica.layout.flowLayout;

import java.awt.FlowLayout;

import javax.swing.JButton;
```

```
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JButton boton1;
   private JButton boton2;
   private JButton boton3;
   private JButton boton4;
   private JButton boton5;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       this.setTitle("Frame Principal");
       FlowLayout thisLayout = new FlowLayout();
       getContentPane().setLayout(thisLayout);
       {
          boton1 = new JButton();
          getContentPane().add(boton1);
          boton1.setText("Boton 1");
       }
       {
          boton2 = new JButton();
          getContentPane().add(boton2);
          boton2.setText("Boton 2");
       }
       {
          boton3 = new JButton();
          getContentPane().add(boton3);
          boton3.setText("Boton 3");
       }
       {
          boton4 = new JButton();
          getContentPane().add(boton4);
          boton4.setText("Boton 4");
       }
       {
          boton5 = new JButton();
          getContentPane().add(boton5);
          boton5.setText("Boton 5");
       }
       setSize(400, 300);
   }
}
```

Al ejecutar la implementación anterior se presenta el resultado de la Figura 44.

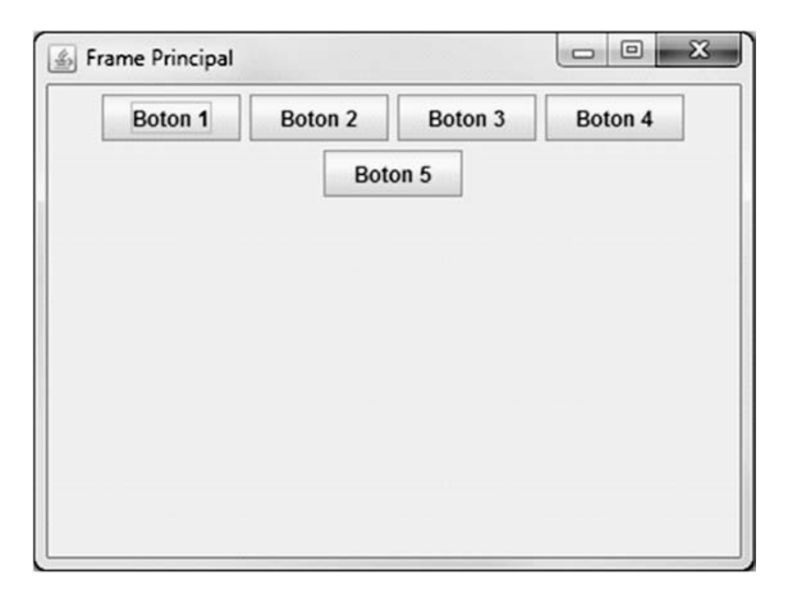


Figura 44. Flow Layout

11.5.4 *GridLayout*

La clase *GridLayout* permite organizar elementos dinámicamente en un contenedor, en donde la posición de los elementos depende del orden en que son agregados al contenedor. Estos elementos están colocados en forma de tabla, con base en filas y columnas definidas en el *GridLayout*. Además, se puede establecer una separación de los componentes agregados en el contenedor.

La sintaxis para asignar un *GridLayout* a un contenedor, como por ejemplo un panel, es la siguiente:

```
JPanel panel = new JPanel();
GridLayout layout = new GridLayout(3, 2, 10, 10);
panel.setLayout(layout);
```

En el ejemplo anterior, el constructor del *GridLayout* recibe cuatro parámetros. El primero, determina el número de filas; el segundo, el

número de columnas; el tercero, la separación de los componentes en el eje X medido en píxeles y el cuarto, la separación de los componentes en el eje Y medido en píxeles.

La siguiente implementación permite establecer un *JFrame* con *GridLayout,* en el cual se adicionan seis botones.

```
package interfazGrafica.layout.gridLayout;

import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JButton boton1;
   private JButton boton2;
   private JButton boton3;
   private JButton boton4;
   private JButton boton5;
   private JButton boton6;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       GridLayout thisLayout = new GridLayout(3, 2, 10, 10);
       getContentPane().setLayout(thisLayout);
       {
          boton1 = new JButton();
          getContentPane().add(boton1);
          boton1.setText("Boton 1");
       }
       {
          boton2 = new JButton();
          getContentPane().add(boton2);
          boton2.setText("Boton 2");
       }
```

```
        {
           boton3 = new JButton();
           getContentPane().add(boton3);
           boton3.setText("Boton 3");
        }
        {
           boton4 = new JButton();
           getContentPane().add(boton4);
           boton4.setText("Boton 4");
        }
        {
           boton5 = new JButton();
           getContentPane().add(boton5);
           boton5.setText("Boton 5");
        }
        {
           boton6 = new JButton();
           getContentPane().add(boton6);
           boton6.setText("Boton 6");
        }
        setSize(400, 300);
    }
}
```

Al ejecutar la implementación anterior, se presenta el resultado de la Figura 45.

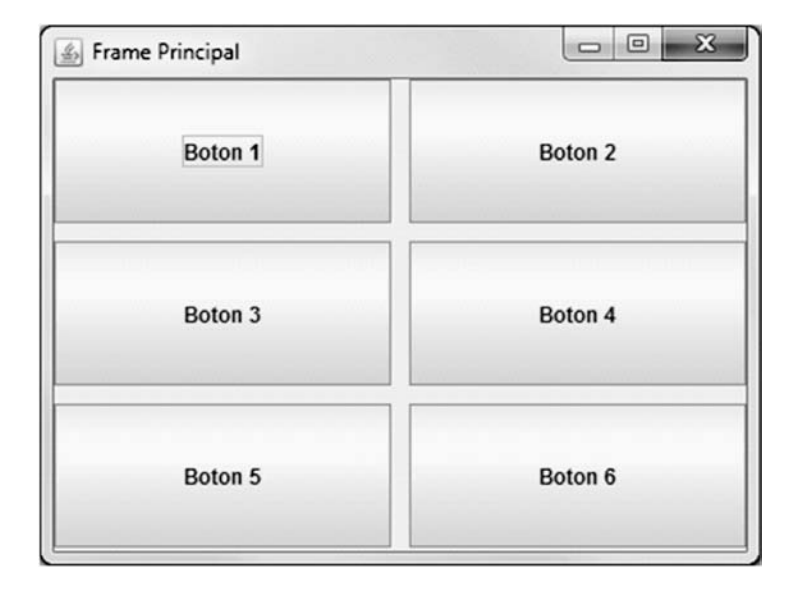


Figura 45. Grid Layout

11.6 Formularios

Con base en contenedores, componentes y *layouts*, se pueden diseñar diferentes tipos de formularios para realizar operaciones típicas de manipulación de datos en sistemas de información.

Por ejemplo, un formulario completo para almacenar información de clientes en un sistema podría contener los siguientes elementos:

- Datos Básicos
    - Identificación
    - Nombre
    - Apellido
    - Género
- Datos de Contacto
    - Correo
    - Teléfono
    - Celular
- Datos de Ubicación
    - Dirección
    - País
    - Departamento
    - Ciudad
- Pasatiempos
    - Deportes
    - *Hobbies*

Con la información anterior se puede hacer un diseño basado en el panel de pestañas, para poder presentar de forma clara todos los elementos para la visualización.

Debido a la gran cantidad de elementos es apropiado crear los paneles de cada pestaña por separado.

La primera pestaña contendrá los datos básicos. Este panel contiene un *GridLayout* de cuatro filas y dos columnas. El género se representa

como *RadioButton*, el cual debe estar dentro de un panel que tiene *FlowLayout*. La implementación del panel correspondiente es la siguiente:

Clase *PDatosBasicos*

```
package interfazGrafica.formulario;

import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridLayout;

import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JTextField;

public class PDatosBasicos extends JPanel {
   private JLabel labelIdentificacion;
   private JTextField textIdentificacion;
   private JLabel labelNombre;
   private JTextField textNombre;
   private JLabel labelApellido;
   private JTextField textApellido;
   private JLabel labelGenero;
   private JPanel panelGenero;
   private ButtonGroup buttonGroupGenero;
   private JRadioButton radioFemenino;
   private JRadioButton radioMasculio;

   public PDatosBasicos() {
       initGUI();
   }

   private void initGUI() {
       setLayout(new GridLayout(4, 2, 5, 5));
       {
          labelIdentificacion = new JLabel();
          this.add(labelIdentificacion);
          labelIdentificacion.setText("Identificacion");
       }
       {
          textIdentificacion = new JTextField();
          this.add(textIdentificacion);
```

```
        }
        {
            labelNombre = new JLabel();
            this.add(labelNombre);
            labelNombre.setText("Nombre");
        }
        {
            textNombre = new JTextField();
            this.add(textNombre);
        }
        {
            labelApellido = new JLabel();
            this.add(labelApellido);
            labelApellido.setText("Apellido");
        }
        {
            textApellido = new JTextField();
            this.add(textApellido);
        }
        {
            labelGenero = new JLabel();
            this.add(labelGenero);
            labelGenero.setText("Genero");
        }
        {
            panelGenero = new JPanel();
            FlowLayout panelGeneroLayout = new FlowLayout();
            this.add(panelGenero);
            panelGenero.setLayout(panelGeneroLayout);
            buttonGroupGenero = new ButtonGroup();
            {
                radioFemenino = new JRadioButton();
                panelGenero.add(radioFemenino);
                radioFemenino.setText("Femenino");
                buttonGroupGenero.add(radioFemenino);
            }
            {
                radioMasculio = new JRadioButton();
                panelGenero.add(radioMasculio);
                radioMasculio.setText("Masculino");
                buttonGroupGenero.add(radioMasculio);
            }
        }
    }
}
```

El resultado de este panel es el siguiente:

Figura 46. Ejemplo de Formulario. Panel datos básicos

La segunda pestaña contendrá los datos de contacto. Este panel contiene un *GridLayout* de tres filas y dos columnas. La implementación del panel correspondiente es la siguiente:

Clase *PDatosContacto*

```
package interfazGrafica.formulario;

import java.awt.Dimension;
import java.awt.GridLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PDatosContacto extends JPanel {
   private JLabel labelCorreo;
   private JTextField textCorreo;
   private JLabel labelCelular;
   private JTextField textCelular;
   private JLabel labelTelefono;
   private JTextField textTelefono;

   public PDatosContacto() {
       initGUI();
   }

   private void initGUI() {
       setLayout(new GridLayout(3, 2, 5, 5));
```

```
        setPreferredSize(new Dimension(400, 150));
        {
           labelCorreo = new JLabel();
           this.add(labelCorreo);
           labelCorreo.setText("Correo");
        }
        {
           textCorreo = new JTextField();
           this.add(textCorreo);
        }
        {
           labelTelefono = new JLabel();
           this.add(labelTelefono);
           labelTelefono.setText("Telefono");
        }
        {
           textTelefono = new JTextField();
           this.add(textTelefono);
        }
        {
           labelCelular = new JLabel();
           this.add(labelCelular);
           labelCelular.setText("Celular");
        }
        {
           textCelular = new JTextField();
           this.add(textCelular);
        }
    }
}
```

El resultado de este panel es el siguiente:

Figura 47. Ejemplo de Formulario. Panel datos de contacto

La tercera pestaña contendrá los datos de ubicación. Este panel contiene un *GridLayout* de cuatro filas y dos columnas. La implementación del panel correspondiente es la siguiente:

Clase *PDatosUbicacion*

```
package interfazGrafica.formulario;

import java.awt.Dimension;
import java.awt.GridLayout;
import javax.swing.ComboBoxModel;
import javax.swing.DefaultComboBoxModel;
import javax.swing.JComboBox;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PDatosUbicacion extends JPanel {
   private JLabel labelDireccion;
   private JTextField textDireccion;
   private JComboBox comboPais;
   private JLabel labelPais;
   private JLabel labelDepartamento;
   private JComboBox comboDepartamento;
   private JLabel labelCiudad;
   private JComboBox comboCiudad;

   public PDatosUbicacion() {
       initGUI();
   }

   private void initGUI() {
       setLayout(new GridLayout(4, 2, 5, 5));
       {
          labelDireccion = new JLabel();
          this.add(labelDireccion);
          labelDireccion.setText("Direccion");
       }
       {
          textDireccion = new JTextField();
          this.add(textDireccion);
       }
       {
          labelPais = new JLabel();
          this.add(labelPais);
          labelPais.setText("Pais");
       }
       {
          ComboBoxModel comboPaisModel =
              new DefaultComboBoxModel(
                 new String[] { "Pais 1", "Pais 2", "Pais 3" });
```

```
            comboPais = new JComboBox();
            this.add(comboPais);
            comboPais.setModel(comboPaisModel);
        }
        {
            labelDepartamento = new JLabel();
            this.add(labelDepartamento);
            labelDepartamento.setText("Departamento");
        }
        {
            ComboBoxModel comboDepartamentoModel =
                new DefaultComboBoxModel(
                    new String[] { "Departamento 1", "Departamento 2" });
            comboDepartamento = new JComboBox();
            this.add(comboDepartamento);
            comboDepartamento.setModel(comboDepartamentoModel);
        }
        {
            labelCiudad = new JLabel();
            this.add(labelCiudad);
            labelCiudad.setText("Ciudad");
        }
        {
            ComboBoxModel comboCiudadModel =
                new DefaultComboBoxModel(
                    new String[] { "Ciudad 1", "Ciudad 2" });
            comboCiudad = new JComboBox();
            this.add(comboCiudad);
            comboCiudad.setModel(comboCiudadModel);
        }
    }
}
```

El resultado de este panel es el siguiente:



Figura 48. Ejemplo de Formulario. Panel datos de ubicación

La cuarta pestaña contendrá pasatiempos. Este panel contiene un *FlowLayout*. La implementación del panel correspondiente es la siguiente:

Clase *PPasatiempos*

```
package interfazGrafica.formulario;

import java.awt.Dimension;
import java.awt.FlowLayout;

import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class PPasatiempos extends JPanel {
   private JCheckBox checkFutbol;
   private JCheckBox checkBaloncesto;
   private JCheckBox checkNatacion;
   private JCheckBox checkCine;
   private JCheckBox checkTeatro;
   private JCheckBox checkAtletismo;
   private JCheckBox checkTenis;

   public PPasatiempos() {
       initGUI();
   }

   private void initGUI() {
       this.setLayout(new FlowLayout());
       setPreferredSize(new Dimension(400, 150));
       {
          checkFutbol = new JCheckBox();
          this.add(checkFutbol);
          checkFutbol.setText("Futbol");
       }
       {
          checkBaloncesto = new JCheckBox();
          this.add(checkBaloncesto);
          checkBaloncesto.setText("Baloncesto");
       }
       {
          checkTenis = new JCheckBox();
          this.add(checkTenis);
          checkTenis.setText("Tenis");
       }
```

```
        {
           checkNatacion = new JCheckBox();
           this.add(checkNatacion);
           checkNatacion.setText("Natacion");
        }
        {
           checkAtletismo = new JCheckBox();
           this.add(checkAtletismo);
           checkAtletismo.setText("Atletismo");
        }
        {
           checkTeatro = new JCheckBox();
           this.add(checkTeatro);
           checkTeatro.setText("Teatro");
        }
        {
           checkCine = new JCheckBox();
           this.add(checkCine);
           checkCine.setText("Cine");
        }
    }
}
```

El resultado de este panel es el siguiente:



Figura 49. Ejemplo de Formulario. Panel pasatiempos

Adicionalmente, un formulario debe tener botones que permitan ejecutar operaciones. Estas operaciones deben ser ejecutadas cuando el usuario lo decida. Entonces, se construye un panel que contenga botones. Este panel contiene un *FlowLayout*. La implementación del panel correspondiente es la siguiente:

PBotones

```
package interfazGrafica.formulario;

import java.awt.Dimension;
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class PBotones extends JPanel {
	private JButton buttonAceptar;
	private JButton buttonCancelar;
	private JButton buttonLimpiar;

	public PBotones() {
		initGUI();
	}

	private void initGUI() {
		setLayout(new FlowLayout());
		setPreferredSize(new Dimension(400, 100));
		{
			buttonAceptar = new JButton();
			this.add(buttonAceptar);
			buttonAceptar.setText("Aceptar");
		}
		{
			buttonCancelar = new JButton();
			this.add(buttonCancelar);
			buttonCancelar.setText("Cancelar");
		}
		{
			buttonLimpiar = new JButton();
			this.add(buttonLimpiar);
			buttonLimpiar.setText("Limpiar");
		}
	}
}
```

El resultado de este panel es el siguiente:



Figura 50. Ejemplo de Formulario. Panel botones

Para incluir los paneles construidos se debe crear un *JFrame* que incluya un *Label* para colocar un título, un *TabbedPane* para incluir los demás paneles en pestañas y un panel de botones. Este *frame* contiene un *BorderLayout*. La implementación es la siguiente:

Clase *FFormulario*

```
package interfazGrafica.formulario;

import java.awt.BorderLayout;

import javax.swing.JLabel;
import javax.swing.JTabbedPane;
import javax.swing.WindowConstants;

public class FFormulario extends javax.swing.JFrame {
   private JLabel labelTitulo;
   private PDatosBasicos panelDatosBasicos;
   private PDatosContacto panelDatosContacto;
   private PDatosUbicacion panelDatosUbicacion;
   private PPasatiempos panelPasatiempos;
   private PBotones panelBotones;
   private JTabbedPane panelPestanas;


   public static void main(String[] args) {
       FFormulario frame = new FFormulario();
       frame.setVisible(true);
   }

   public FFormulario() {
       initGUI();
       labelTitulo = new JLabel();
       labelTitulo.setText("Formulario Usuario");
       labelTitulo.setHorizontalAlignment(JLabel.CENTER);
       panelDatosBasicos = new PDatosBasicos();
       panelDatosContacto = new PDatosContacto();
       panelDatosUbicacion = new PDatosUbicacion();
       panelPasatiempos = new PPasatiempos();
       panelBotones = new PBotones();
       panelPestanas = new JTabbedPane();
       getContentPane().add(labelTitulo, BorderLayout.NORTH);
       getContentPane().add(panelPestanas, BorderLayout.CENTER);
```

```
        {
           panelPestanas.addTab(
           "Datos Basicos", panelDatosBasicos);
           panelPestanas.addTab(
           "Datos Contacto", panelDatosContacto);
           panelPestanas.addTab(
           "Datos Ubicacion", panelDatosUbicacion);
           panelPestanas.addTab("Pasatiempos", panelPasatiempos);
        }
        getContentPane().add(panelBotones, BorderLayout.SOUTH);
    }

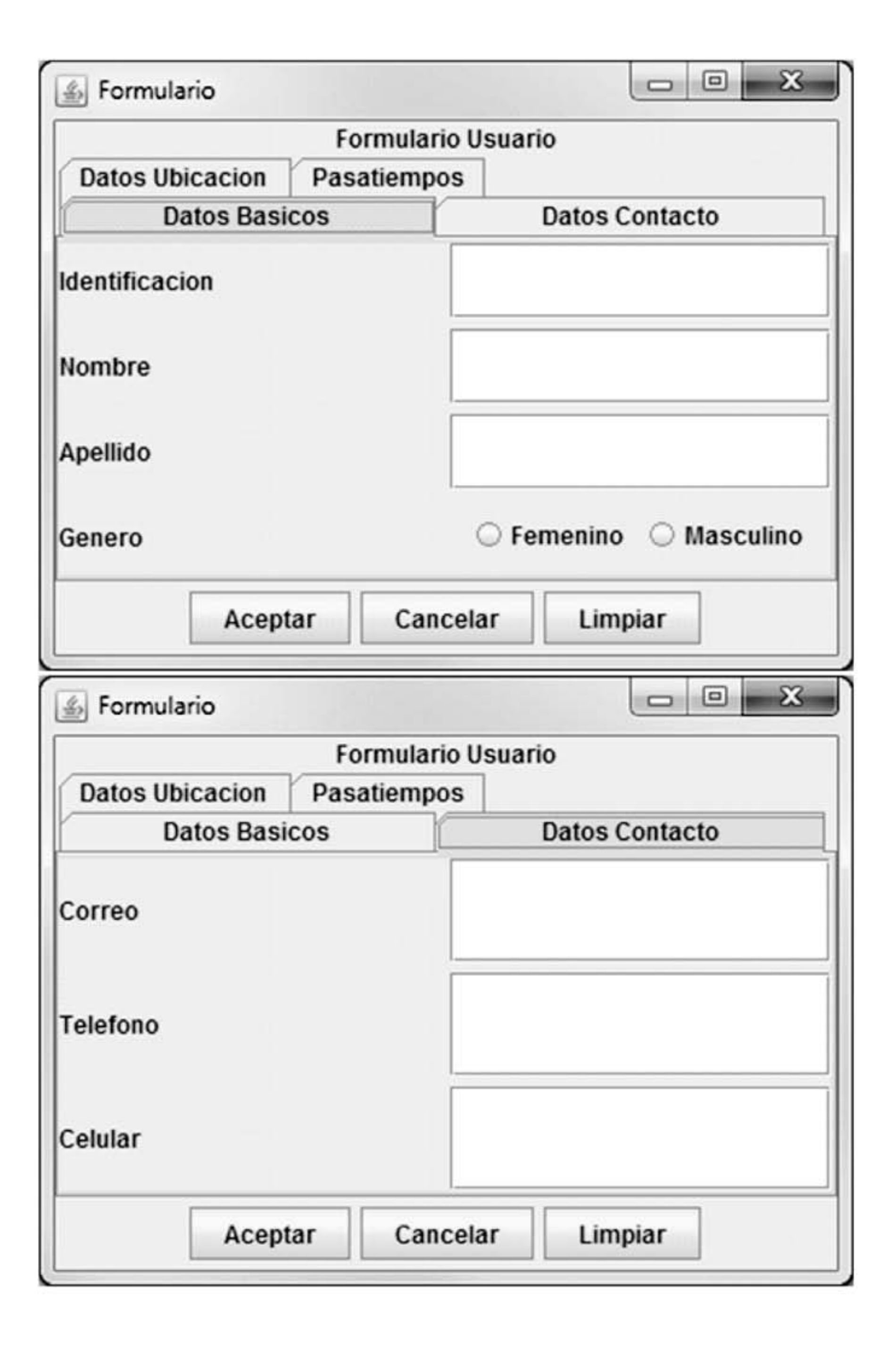
    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setLayout(new BorderLayout());
        setTitle("Formulario");
        setSize(400, 300);
    }
}
```

En la implementación se pueden apreciar los siguientes detalles:

- El *frame* contiene un *border layout* en el que se agregan, un panel de título en el norte, un panel de pestañas en el centro y un panel de botones en el sur.
- El panel de botones se configuraron *flow layout* con el fin de que quedaran organizados en el centro y linealmente.
- La información en el panel de pestañas se ha dividido por medio de paneles independientes los cuales contienen los datos básicos, datos de contacto, datos de ubicación, los cuales están configurados con *grid layout* y un panel de datos de pasatiempos, el cual está configurado con *flow layout.*

Esta organización de elementos permite una cómoda visualización y proporciona un gran conjunto de atributos para la manipulación del usuario.

Los resultados de esta implementación en cada una de las pestañas se presentan en la Figura 51.


Formulario
Formulario Usuario
Datos Ubicacion
Pasatiempos
Datos Basicos
Datos Contacto
Identificacion
Nombre
Apellido
Genero
Femenino
Masculino
Aceptar
Cancelar
Limpiar
Formulario
Formulario Usuario
Datos Ubicacion
Pasatiempos
Datos Basicos
Datos Contacto
Correo
Telefono
Celular
Aceptar
Cancelar
Limpiar

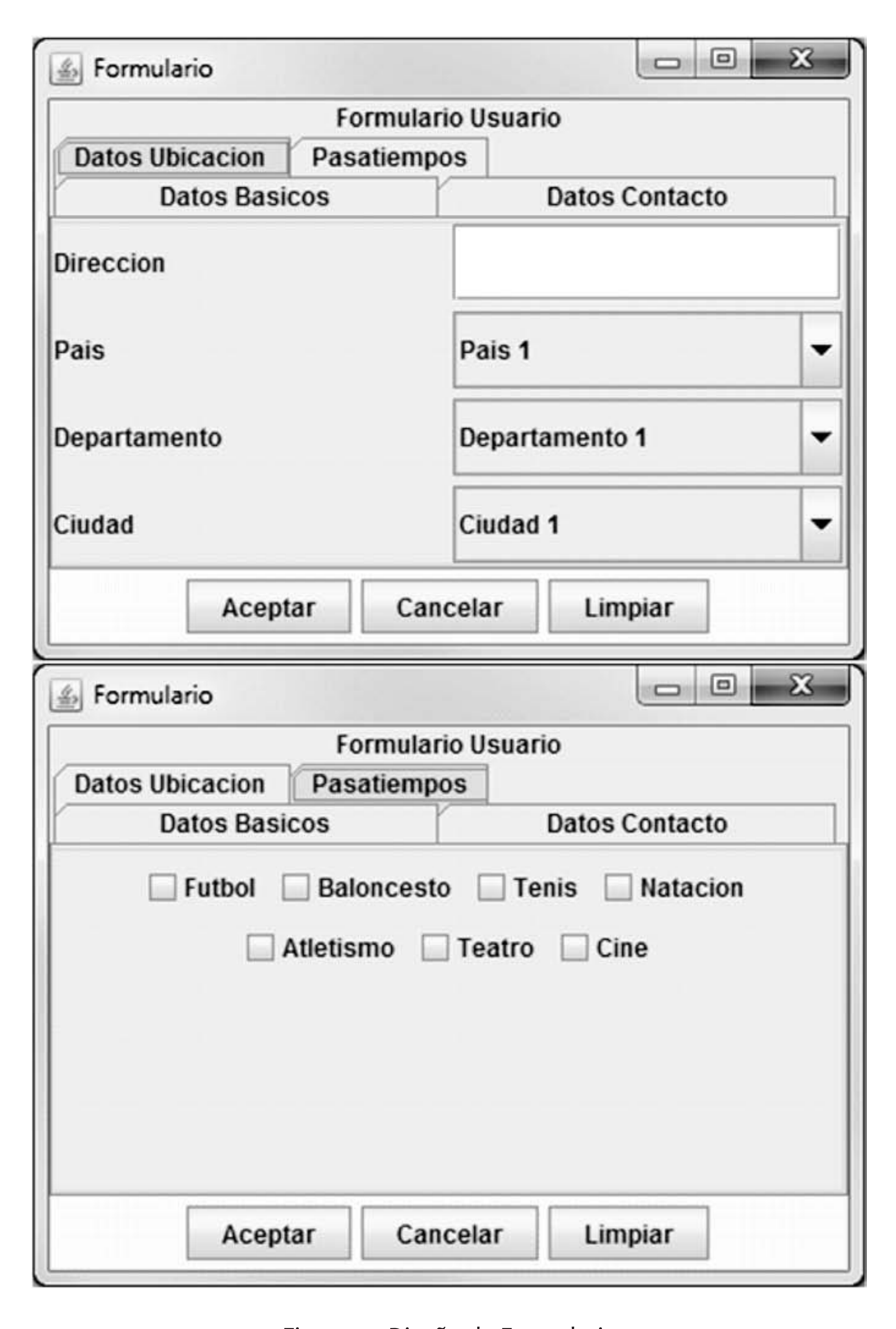


Figura 51. Diseño de Formulario

11.7 Manejo de eventos

Los eventos en Java son posibles gracias a las interfaces que definen los comportamientos necesarios, para cada uno de los componentes de acuerdo a sus características.

11.7.1 *ActionListener*

La interfaz *ActionListener* permite ejecutar un método denominado *actionPerformed,* en el momento en que se da un evento de clic sobre un componente. El componente recibe la implementación del *ActionListener* mediante el método *addActionListener*. Es necesario implementar el método *actionPerformed,* el cual recibe un parámetro *ActionEvent* que contiene información del componente que ha invocado el evento. Este método se encuentra disponible en las clases *JButton*, *JRadioButton*, *JCheckBox*, *JComboBox* y *JTextField*.

Con base en un botón, la sintaxis para implementar un evento *actionPerformed* es la siguiente:

```
JButton boton = new JButton();
boton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {
       botonActionPerformed(evt);
   }
});

private void botonActionPerformed(ActionEvent evt) {
   //Código para el evento
}
```

11.7.2 *KeyListener*

La interfaz *KeyListener* permite ejecutar tres métodos denominados *keyTyped*, *keyReleased* y *keyPressed*. El método *keyTyped* se invoca cuando se digita, es decir, oprime y suelta una tecla; el método *keyRealeased* se invoca cuando se suelta una tecla y el método *keyPressed* se invoca cuando se presiona una tecla. El componente

recibe la implementación del *KeyListener* a través de la instancia de la clase *KeyAdapter*, mediante el método *addKeyListener*. Es necesario implementar el método *keyType*, *keyReleased* y *keyPressed;* los cuales reciben un parámetro *KeyEvent* que contiene información del componente que ha invocado el evento. Este método se encuentra disponible en las clases *JButton*, *JRadioButton*, *JCheckBox*, *JComboBox*, *JLabel*, *JList*, *JTextField*, *JTextArea*, *JTable* y JPanel.

Con base en un cuadro de texto, la sintaxis para implementar los eventos *keyTyped*, *keyReleased* y *keyPressed* es la siguiente:

```
JTectField cuadroTexto = new JTextField();
cuadroTexto.addKeyListener(new KeyAdapter() {
   public void keyTyped(KeyEvent evt) {
       cuadroTextoKeyTyped(evt);
   }
   public void keyReleased(KeyEvent evt) {
       cuadroTextoKeyReleased(evt);
   }
   public void keyPressed(KeyEvent evt) {
       cuadroTextoKeyPressed(evt);
   }
});

private void cuadroTextoKeyPressed(KeyEvent evt) {
   //Código para el evento
}

private void cuadroTextoKeyReleased(KeyEvent evt) {
   //Código para el evento
}

private void cuadroTextoKeyTyped(KeyEvent evt) {
       //Código para el evento
}
```

11.7.3 *FocusListener*

La interfaz *FocusListener* permite ejecutar dos métodos denominados *focusLost* y *focusGained*. El método *focusLost* se invoca cuando un componente pierde el foco, es decir, cuando este componente deja

de estar seleccionado y el método *focusGained* se invoca cuando un componente adquiere el foco, es decir, cuando este componente se selecciona. El componente recibe la implementación del *FocusListener* a través de la instancia de la clase *FocusAdapter* mediante el método *addFocusListener*. Es necesario implementar el método *focusLost* y *focusGained*, los cuales reciben un parámetro *FocusEvent* que contiene información del componente que ha invocado el evento. Este método se encuentra disponible en todos los componentes de *Swing*.

Con base en un cuadro de texto, la sintaxis para implementar los eventos *focusLost* y *focusGained* es la siguiente:

```
cuadroTexto = new JTextField();
cuadroTexto.addFocusListener(new FocusAdapter() {
   public void focusLost(FocusEvent evt) {
       cuadroTextoFocusLost(evt);
   }
   public void focusGained(FocusEvent evt) {
       cuadroTextoFocusGained(evt);
   }
});

private void cuadroTextoFocusGained(FocusEvent evt) {
   //Código para el evento
}

private void cuadroTextoFocusLost(FocusEvent evt) {
   //Código para el evento
}
```

11.7.4 *MouseListener*

La interfaz *MouseListener* permite ejecutar tres métodos denominados *mouseClicked*, *mouseReleased* y *mousePressed*. El método *mouseClicked* se invoca cuando se oprime y suelta un botón del *mouse*, el método *mouseReleased* se invoca cuando se suelta un botón del *mouse* y el método *mousePressed* se invoca cuando se presiona un botón del *mouse*. El componente recibe la implementación del *MouseListener* a través de la instancia de la clase *MouseAdapter* mediante el método *addMouseListener*. Es necesario implementar el método *mouseClicked*, *mouseReleased* y *mousePressed*; los cuales reciben un

parámetro *MouseEvent* que contiene información del componente que ha invocado el evento. Este método se encuentra disponible en todos los componentes de *Swing*.

Con base en un panel, la sintaxis para implementar los eventos *mouseClicked*, *mouseReleased* y *mousePressed* es la siguiente:

```
JPanel panel = new JPanel();
panel.addMouseListener(new MouseAdapter() {
   public void mouseReleased(MouseEvent evt) {
       panelMouseReleased(evt);
   }
   public void mousePressed(MouseEvent evt) {
       panelMousePressed(evt);
   }
   public void mouseClicked(MouseEvent evt) {
       panelMouseClicked(evt);
   }
});

private void panelMouseClicked(MouseEvent evt) {
   //Código para el evento
}

private void panelMousePressed(MouseEvent evt) {
   //Código para el evento
}

private void panelMouseReleased(MouseEvent evt) {
   //Código para el evento
}
```

11.7.5 *MouseMotionListener*

La interfaz *MouseMotionListener* permite ejecutar dos métodos denominados *mouseDragged* y *mouseMoved*. El método *mouseDragged* se invoca cuando se arrastra el puntero del *mouse* sobre el componente, esto quiere decir, que se mueve el puntero mientras está oprimido cualquier botón del *mouse* y el método *mouseMoved* se invoca cuando se mueve el puntero del *mouse* sobre el componente. El componente recibe la implementación del *MouseMotionListener* a través de la instancia de la clase *MouseMotionAdapter,* mediante el método *addMouseMotionListener*. Es necesario implementar

el método *mouseDragged* y *mouseMoved*, los cuales reciben un parámetro *MouseEvent* que contiene información del componente que ha invocado el evento. Este método se encuentra disponible en todos los componentes de *Swing*.

Con base en un panel, la sintaxis para implementar los eventos *mouseDragged* y *mouseMoved* es la siguiente:

```
JPanel panel = new JPanel();
panel.addMouseMotionListener(new MouseMotionAdapter() {
   public void mouseMoved(MouseEvent evt) {
       panelMouseMoved(evt);
   }
   public void mouseDragged(MouseEvent evt) {
       panelMouseDragged(evt);
   }
});

private void panelMouseDragged(MouseEvent evt) {
   //Código para el evento
}

private void panelMouseMoved(MouseEvent evt) {
   //Código para el evento
}
```

11.8 Menús

Los menús en Java se comportan de forma similar a los botones debido a que poseen la misma clase de eventos. Para crear una barra de menú funcional es necesario crear una barra de menú a través de la clase *JMenuBar*. La barra de menú se compone de *JMenu,* los cuales son menús que no deben contener eventos sino que permite desplegar *JMenuItem*, que permiten implementar eventos para proporcionar servicios a la aplicación. Existen otro tipo de menús que pueden ser utilizados para maximizar la funcionalidad de la aplicación como *JCheckBoxMenuItem* y *JRadioButtonMenuItem*.

Java también permite la creación de menús flotantes a través de *JPopUpMenu*, el cual posee las mismas ventajas al *JMenuBar*.

11.8.1 *JMenuBar*

El *JMenuBar* proporciona una barra de menú, la cual se comporta como un contenedor de menús. Una barra de menú, necesariamente, debe agregarse a un *JFrame*. La sintaxis para crear y asignar una barra de menú es la siguiente:

```
JMenuBar menuBar = new JMenuBar();
this.setJMenuBar(menuBar);
```

Donde el apuntador *this* hace referencia al *JFrame* en donde se crea la barra de menú.

11.8.2 *JMenu, JMenuItem* y *JMenuSeparator*

El *JMenu* proporciona un menú, el cual puede contener más, *JMenu* o *JMenuItem*. El menú debe agregarse a otro menú o a una barra de menú a través del método *add*.

Un *JMenu* puede contener un ícono, el cual puede ser una imagen con extensión *jpg*, *gif*, *png* o *ico*. La imagen debe encontrarse en una carpeta dentro del proyecto. Para asignar la imagen a un menú es necesario usar el método *setIcon,* en el cual se envía como parámetro la sentencia *getClass().getClassLoader().getResource*(“img/nuevo.*png*”)). En este caso, el ícono se denomina nuevo.*png*.

La sintaxis para crear y asignar un menú es la siguiente:

```
JMenu menuArchivo = new JMenu();
barraMenu.add(menuArchivo);
menuArchivo.setText(“Archivo”);
JMenu menuNuevo = new JMenu();
mArchivo.add(menuNuevo);
menuNuevo.setText(“Nuevo”);
menuNuevo.setIcon(new ImageIcon(“img/ayuda.png”));
```

Donde *menuNuevo* está contenido en *menuArchivo* y este, en la barra de menú. Además, *menuNuevo* contiene un ícono que acompaña al texto del menú.

El *JMenuItem* proporciona un menú final, el cual puede ejecutar servicios. El menú *ítem* debe agregarse necesariamente a un menú, a través del método *add*.

La sintaxis para crear y asignar un menú *ítem* e implementar un evento para el menú *ítem* es la siguiente:

```
JMenuItem menuItemArchivoSecuencial = new JMenuItem();
mNuevo.add(menuItemArchivoSecuencial);
menuItemArchivoSecuencial.setText("Archivo Secuencial");
menuItemArchivoSecuencial.addActionListener(new ActionListener()
{
   public void actionPerformed(ActionEvent evt) {
       menuItemArchivoSecuencialActionPerformed(evt);
   }
});

private void menuItemArchivoSecuencialActionPerformed(
ActionEvent evt) {
   //Código para el evento
}
```

El *JSeparator* permite colocar una línea de separación entre menús. La sintaxis para crear y asignar un separador es la siguiente:

```
JSeparator separador1 = new JSeparator();
menuArchivo.add(separador1);
```

El siguiente ejemplo implementa una barra de menú con diferentes menús y menú *ítems*.

```
package interfazGrafica.menu;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
```

```
import javax.swing.JSeparator;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JMenuBar menuBar;
   private JMenu menuAyuda;
   private JMenu menuNuevo;
   private JMenuItem menuItemCerrar;
   private JSeparator separador1;
   private JMenuItem menuItemArchivoSerializable;
   private JMenuItem menuItemArchivoSecuencial;
   private JMenu menuContenido;
   private JMenu menuAbrir;
   private JMenu menuArchivo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       this.setTitle("Frame Principal");
       {
          menuBar = new JMenuBar();
          setJMenuBar(menuBar);
          {
              menuArchivo = new JMenu();
              menuBar.add(menuArchivo);
              menuArchivo.setText("Archivo");
              {
                 menuNuevo = new JMenu();
                 menuArchivo.add(menuNuevo);
                 menuNuevo.setText("Nuevo");
                 {
                     menuItemArchivoSecuencial = new JMenuItem();
                     menuNuevo.add(menuItemArchivoSecuencial);
                     menuItemArchivoSecuencial.setText(
                     "Archivo Secuencial");
                     menuItemArchivoSecuencial.addActionListener(
                     new ActionListener() {
                        public void actionPerformed(
                        ActionEvent evt) {
                            menuItemArchivoSecuencialAction
```

```
                        Performed(evt);
                    }
                });
            }
            {
                menuItemArchivoSerializable =
                new JMenuItem();
                menuNuevo.add(menuItemArchivoSerializable);
                menuItemArchivoSerializable.setText(
                "Archivo Serializable");
            }
        }
        {
            menuAbrir = new JMenu();
            menuArchivo.add(menuAbrir);
            menuAbrir.setText("Abrir");
        }
        {
            separador1 = new JSeparator();
            menuArchivo.add(separador1);
        }
        {
            menuItemCerrar = new JMenuItem();
            menuArchivo.add(menuItemCerrar);
            menuItemCerrar.setText("Cerrar");
        }
    }
    {
        menuAyuda = new JMenu();
        menuBar.add(menuAyuda);
        menuAyuda.setText("Ayuda");
        menuAyuda.setIcon(new ImageIcon("img/ayuda.png"));
        {
            menuContenido = new JMenu();
            menuAyuda.add(menuContenido);
            menuContenido.setText("Contenido");
        }
    }
  }
  setSize(400, 300);
 }

   private void menuItemArchivoSecuencialActionPerformed(ActionE
vent evt) {
       //Código para el evento del menu item
   }
}
```

El resultado es el siguiente:



Figura 52. *JMenuBar*, *JMenu* y *JMenuItem*

En la figura anterior se puede apreciar que el ícono del menú *Ayuda* se ve bastante bien. Con base en ello se recomienda que las imágenes de los menús tengan una resolución de 16 x 16 píxeles.

11.8.3 *JCheckBoxMenuItem* y *JRadioButtonMenuItem*

El *JCheckBoxMenuItem* permite colocar un *CkeckBox* en un menú. Esta característica es muy típica en menús que permitan la visualización particular de algún componente, como la barra de herramientas o la barra de estado. La sintaxis para crear y asignar un *JCheckBoxMenuItem* es la siguiente:

```
JCheckBoxMenuItem checkMenuItemBarraEstado = new
JCheckBoxMenuItem();
menu.add(checkMenuItemBarraEstado);
checkMenuItemBarraEstado.setText("Barra de Estado");
```

El *JRadioButtonMenuItem* permite colocar un *Button* en un menú. Esta característica es muy típica en menús que permitan la

visualización particular de algún componente, como la barra de herramientas o la barra de estado. La sintaxis para crear y asignar un *JCheckBoxMenuItem* es la siguiente:

```
JRadioButtonMenuItem radioMenuItemVistaMiniatura = new
JRadioButtonMenuItem();
menu.add(radioMenuItemVistaMiniatura);
radioMenuItemVistaMiniatura.setText("Vista en miniatura");
```

El siguiente ejemplo implementa una barra de menú con diferentes menús *ítems*.

```
package interfazGrafica.menu;

import javax.swing.ButtonGroup;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JSeparator;
import javax.swing.WindowConstants;

public class FPrincipal2 extends javax.swing.JFrame {
   private JMenuBar menuBar;
   private JMenu menuVer;
   private ButtonGroup buttonGroup;
   private JRadioButtonMenuItem radioMenuItemLista;
   private JRadioButtonMenuItem radioMenuItemIconos;
   private JRadioButtonMenuItem radioMenuItemMosaico;
   private JRadioButtonMenuItem radioMenuItemVistaMiniatura;
   private JSeparator separador1;
   private JCheckBoxMenuItem checkMenuItemBarraEstado;

   public static void main(String[] args) {
       FPrincipal2 frame = new FPrincipal2();
       frame.setVisible(true);
   }

   public FPrincipal2() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       this.setTitle("Frame Principal");
       getContentPane().setLayout(null);
       {
          menuBar = new JMenuBar();
```

```
            setJMenuBar(menuBar);
            {
                menuVer = new JMenu();
                menuBar.add(menuVer);
                menuVer.setText("Ver");
                {
                   checkMenuItemBarraEstado =
                   new JCheckBoxMenuItem();
                   menuVer.add(checkMenuItemBarraEstado);
                   checkMenuItemBarraEstado.setText(
                   "Barra de Estado");
                }
                {
                   separador1 = new JSeparator();
                   menuVer.add(separador1);
                }
                {
                   radioMenuItemVistaMiniatura =
                   new JRadioButtonMenuItem();
                   menuVer.add(radioMenuItemVistaMiniatura);
                   radioMenuItemVistaMiniatura.setText(
                   "Vista en miniatura");
                }
                {
                   radioMenuItemMosaico =
                   new JRadioButtonMenuItem();
                   menuVer.add(radioMenuItemMosaico);
                   radioMenuItemMosaico.setText("Mosaico");
                }
                {
                   radioMenuItemIconos =
                   new JRadioButtonMenuItem();
                   menuVer.add(radioMenuItemIconos);
                   radioMenuItemIconos.setText("Iconos");
                }
                {
                   radioMenuItemLista = new JRadioButtonMenuItem();
                   menuVer.add(radioMenuItemLista);
                   radioMenuItemLista.setText("Lista");
                }
                buttonGroup = new ButtonGroup();
                buttonGroup.add(radioMenuItemVistaMiniatura);
                buttonGroup.add(radioMenuItemMosaico);
                buttonGroup.add(radioMenuItemIconos);
                buttonGroup.add(radioMenuItemLista);
            }
        }
        setSize(400, 300);
    }
}
```

El resultado es el siguiente:

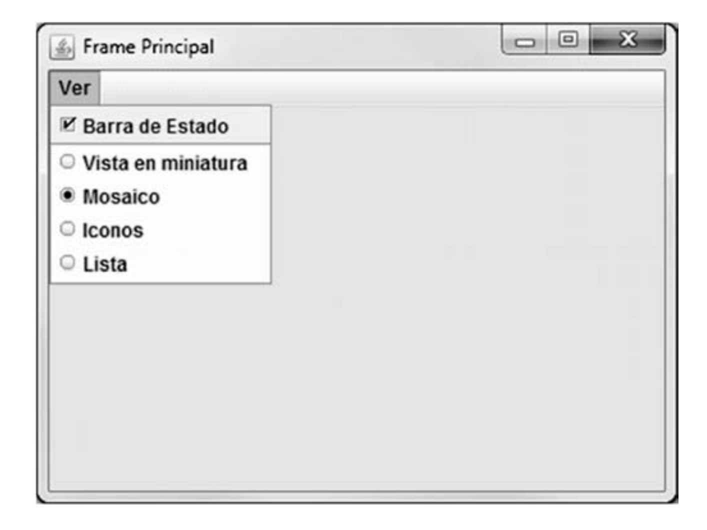


Figura 53. *JMenuBar*, *JCheckBoxMenuItem* y *JRadioButtonMenuItem*

11.8.4 JPopUpMenu

El *JPopUpMenu* permite la creación de menús emergentes que serán visualizados al hacer clic con el botón secundario del *mouse*. Para que esta funcionalidad se presente es necesario hacer uso de la interfaz *mouseListener*, para poder crear los métodos *mousePressed* y *mouseReleased*. La sintaxis para crear y asignar un *JPopUpMenu* es la siguiente:

```
JPopupMenu popUp = new JPopupMenu();
setComponentPopupMenu(this, popUp);
```

El método *setComponentPopupMenu* es el siguiente:

```
package interfazGrafica.menu;
import java.awt.Component;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;
```

```
import javax.swing.WindowConstants;

public class FPrincipal3 extends JFrame {
   private JPopupMenu popUpMenu;
   private JMenuItem menuItemCopiar;
   private JMenuItem menuItemPegar;
   private JMenuItem menuItemCortar;

   public static void main(String[] args) {
       FPrincipal3 frame = new FPrincipal3();
       frame.setVisible(true);
   }

   public FPrincipal3() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       {
          popUpMenu = new JPopupMenu();
          setComponentPopupMenu(this, popUpMenu);
          {
              menuItemCopiar = new JMenuItem();
              popUpMenu.add(menuItemCopiar);
              menuItemCopiar.setText("Copiar");
          }
          {
              menuItemPegar = new JMenuItem();
              popUpMenu.add(menuItemPegar);
              menuItemPegar.setText("Pegar");
          }
          {
              menuItemCortar = new JMenuItem();
              popUpMenu.add(menuItemCortar);
              menuItemCortar.setText("Cortar");
          }
       }
       setSize(400, 300);
   }

   private void setComponentPopupMenu(final Component parent,
final JPopupMenu menu) {
       parent.addMouseListener(new MouseAdapter() {
          public void mousePressed(MouseEvent e) {
              if(e.isPopupTrigger())
```

```
                menu.show(parent, e.getX(), e.getY());
            }
            public void mouseReleased(MouseEvent e) {
                if(e.isPopupTrigger())
                    menu.show(parent, e.getX(), e.getY());
            }
        });
    }
}
```

El resultado es el siguiente:

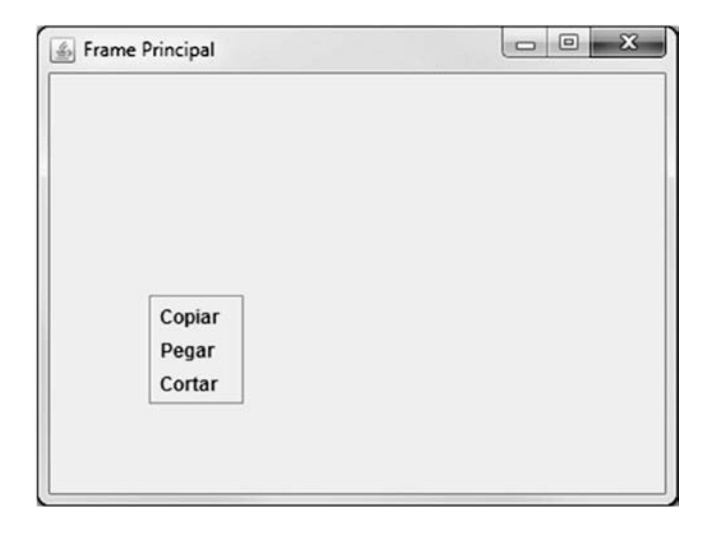


Figura 54. *PopUpMenu*

11.9 *Applets*

Un *Applet* es un contenedor similar a un *JFrame* con una gran variedad de aplicaciones. En un *Applet* se pueden realizar aplicaciones con todos los contenedores que se pueden usar en un *JFrame*. Los *Applets* tienen una característica adicional al *JFrame* que consiste en qué puede ser visualizado en una página *web* a través de lenguaje *HTML*. La máquina virtual de Java cuenta con una aplicación denominada "*Applet Viewer*", la cual permite visualizar el *Applet* como una aplicación de escritorio.

La sintaxis para implementar un *Applet* es la siguiente:

```
package interfazGrafica.applet;

import javax.swing.JApplet;

public class MiApplet extends JApplet {

   public MiApplet() {
       initGUI();
   }

   private void initGUI() {
       //TODO codigo del Applet
   }
}
```

Al ejecutar el *Applet* se inicia la aplicación de Java *Applet Viewer,* presentando el resultado de la Figura 55.

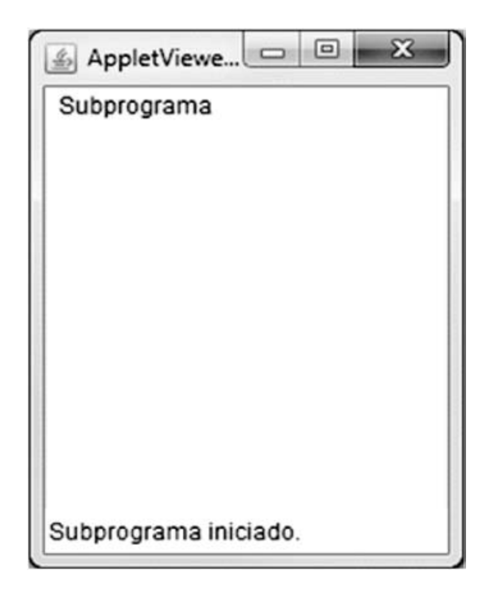


Figura 55. Applet Viewer

Al agregar algunos componentes se puede obtener una aplicación que pueda ser publicada en la *web*.

La siguiente implementación presenta el factorial de un número ingresado, a través de un cuadro de texto.

```
package interfazGrafica.applet;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

public class Applet extends JApplet {
   private JLabel labelTitulo;
   private JButton botonCalcular;
   private JPanel panel;
   private JTextField textoNumero;

   public Applet() {
       initGUI();
   }

   private void initGUI() {
       getContentPane().setLayout(new BorderLayout());
       {
          labelTitulo = new JLabel();
          getContentPane().add(labelTitulo, BorderLayout.NORTH);
          labelTitulo.setText("Este es un Applet");
          labelTitulo.setFont(new Font("Tahoma",1,16));
          labelTitulo.setHorizontalAlignment(
          SwingConstants.CENTER);
       }
       {
          panel = new JPanel();
          FlowLayout panelLayout = new FlowLayout();
          getContentPane().add(panel, BorderLayout.CENTER);
          panel.setLayout(panelLayout);
          {
              textoNumero = new JTextField();
              panel.add(textoNumero);
              textoNumero.setPreferredSize(new Dimension(
              100, 20));
          }
          {
```

```
                botonCalcular = new JButton();
                panel.add(botonCalcular);
                botonCalcular.setText("Calcular Factorial");
                botonCalcular.addActionListener(
                new ActionListener() {
                   public void actionPerformed(ActionEvent evt) {
                      botonCalcularActionPerformed(evt);
                   }
                });
             }
          }
       }

       private void botonCalcularActionPerformed(ActionEvent evt) {
          JOptionPane.showMessageDialog(this, "El factorial de "+
          this.textoNumero.getText()+" es: "+
          factorial(Integer.parseInt(this.textoNumero.getText())),
          "Mensaje",JOptionPane.INFORMATION_MESSAGE);
       }

       private int factorial(int n){
          return (n==1)?1:n*factorial(n-1);
       }
    }
```

El resultado es el siguiente:

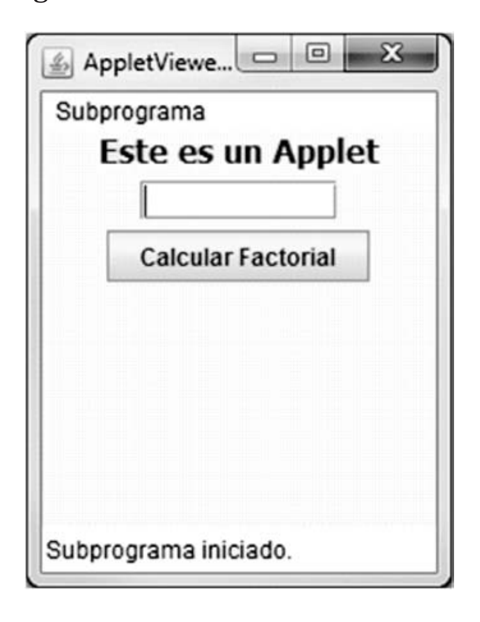


Figura 56. Applet con componentes

Al colocar el número 5 y dar clic en el botón "*Calcular Factorial*", el resultado es el siguiente:



Figura 57. Applet con componentes y cuadro de diálogo

Para visualizar el *Applet* en una página *web*, se hace uso del *tag* "*<applet></applet>*" en donde se incluye el archivo "***.class***", generado por Java a través del atributo "*code*". A este *tag* también se le puede agregar diferentes atributos que definan propiedades físicas del *Applet* como "*width*" para definir el ancho en píxeles y "h*eight*" para definir el alto en píxeles. La implementación es la siguiente:

```
<html>
   <head>
       <title>Publicacion de Applet</title>
   </head>
   <body>
       <h3 align="center"
       >APPLET PARA CALCULAR EL FACTORIAL PUBLICADO CON HTML</h3>
       <div align="center">
          <applet code="Applet.class" height="200" width="200"
          border="2"></applet>
       </div>
   </body>
</html>
```

El resultado presentado en un explorador de Internet es el siguiente:



Figura 58. Applet publicado en página web

Para lograr el resultado anterior es necesario que el archivo "**Applet.class**" se encuentre en la misma ubicación del archivo "**Applet.html**". Al colocar un dato, hacer clic en el botón "*Calcular Factorial*", el resultado es el siguiente:



Figura 59. Applet publicado en página web con cuadro de diálogo

En caso de que se requiera publicar un *applet* que usa clases, es necesario exportar todas las clases relacionadas a un archivo *JAR*. Por ejemplo, la Figura 59 podría ser implementada en dos clases diferentes que se encuentran en paquetes diferentes. La implementación es la siguiente:

Clase *Applet*

```
package interfazGrafica.applet;

import interfazGrafica.applet.util.Matematicas;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

public class Applet extends JApplet {
   private JLabel labelTitulo;
   private JButton botonCalcular;
   private JPanel panel;
   private JTextField textoNumero;

   public Applet() {
      initGUI();
   }

   private void initGUI() {
      getContentPane().setLayout(new BorderLayout());
      {
         labelTitulo = new JLabel();
         getContentPane().add(labelTitulo, BorderLayout.NORTH);
         labelTitulo.setText("Este es un Applet");
         labelTitulo.setFont(new Font("Tahoma",1,16));
         labelTitulo.setHorizontalAlignment(
         SwingConstants.CENTER);
      }
      {
         panel = new JPanel();
         FlowLayout panelLayout = new FlowLayout();
         getContentPane().add(panel, BorderLayout.CENTER);
         panel.setLayout(panelLayout);
         {
            textoNumero = new JTextField();
            panel.add(textoNumero);
            textoNumero.setPreferredSize(new Dimension(100, 20));
         }
         {
            botonCalcular = new JButton();
            panel.add(botonCalcular);
```

```
            botonCalcular.setText("Calcular Factorial");
            botonCalcular.addActionListener(
            new ActionListener() {
               public void actionPerformed(ActionEvent evt) {
                  botonCalcularActionPerformed(evt);
               }
            });
         }
      }
   }

   private void botonCalcularActionPerformed(ActionEvent evt) {
      JOptionPane.showMessageDialog(this, "El factorial de
      "+ this.textoNumero.getText()+" es:
      "+ Matematicas.factorial(Integer.parseInt(
      this.textoNumero.getText())), "Mensaje",
      JOptionPane.INFORMATION_MESSAGE);
   }
}
```

Clase *Matematicas*

```
package interfazGrafica.applet.util;

public class Matematicas {

   public static int factorial(int n){
      return (n==1)?1:n*factorial(n-1);
   }
}
```

Archivo *Applet.html*

```
<html>
   <head>
      <title>Publicacion de Applet</title>
   </head>
   <body>
      <h3 align="center"
      >APPLET PARA CALCULAR EL FACTORIAL PUBLICADO CON HTML</h3>
      <div align="center">
         <applet archive="Applet.jar" code=
         "interfazGrafica.applet.Applet.class" height=
         "200" width="200" border="2"></applet>
      </div>
   </body>
</html>
```

El resultado obtenido en este ejemplo es el mismo que en el caso anterior. Para lograr el resultado es necesario que el archivo "*Applet.jar*" se encuentre en la misma ubicación del archivo "*Applet.html*".

11.10 Ejercicios propuestos

1. Implemente una aplicación *MDI,* aplicando arquitectura de tres capas, que contenga un formulario para ingresar información, uno para consultar y otro que permita visualizar un conjunto de datos en una tabla. Esta información debe encontrarse en un vector. El acceso a los diferentes formularios deben realizarse con base en menús.

2. Implemente una aplicación aplicando arquitectura de tres capas que permita almacenar un vector en un archivo serializable con base en *JFileChooser*.

3. Implemente un *applet* aplicando arquitectura de tres capas que permita almacenar un vector en un archivo serializable con base en *JFileChooser*.

4. Implemente una aplicación utilizando arquitectura de tres capas que permita almacenar y consultar información de un archivo secuencial, aplicando un *InternalFrame* para almacenar y otro *InternalFrame* para consultar.

CAPÍTULO 12

Gráficos

Los gráficos en Java permiten realizar cualquier tipo de dibujo con base en figuras básicas como rectángulos, arcos, imágenes, textos, óvalos, polígonos y polilíneas.

Un gráfico se puede realizar en cualquier contenedor, sin embargo, la mejor práctica es realzarlo sobre un *JPanel*. El *JPanel* contiene un método sobre escribible denominado *Paint*, que recibe un parámetro *Graphics*. En este método se debe realizar toda la implementación de dibujo de la aplicación.

Para asegurar el bajo acoplamiento se recomienda aplicar una arquitectura, en donde, en su capa de presentación se cree una clase que herede de *JPanel y* en donde se implementen los diferentes algoritmos de dibujo. En este caso se debe incluir un *JFrame* y en allí incluir una referencia del *JPanel* que se agregaría al *JFrame*. Para agregar el *JPanel* al *JFrame* se hace necesario especificar un *Layout* al *JFrame*. Si no se le asigna *Layout*, por defecto, tendrá *null*, el cual permitiría agregar el *JPanel* con un tamaño y posición fija.

Lo más apropiado es agregar al *JFrame Border Layout*, de esa forma el *JPanel* quedará con tamaño dinámico, pero el dibujo tendrá que ser también dinámico. La implementación del *JFrame* es la siguiente:

Clase *PDibujo*

```
package graficas;

import java.awt.Graphics;
import javax.swing.JPanel;
```

```
public class PDibujo extends javax.swing.JPanel {

   public PDibujo() {

   }

   public void paint(Graphics g){

   }
}
```

Clase *FPrincipal*

```
package graficas;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       SwingUtilities.invokeLater(new Runnable() {
          public void run() {
              FPrincipal inst = new FPrincipal();
              inst.setLocationRelativeTo(null);
              inst.setVisible(true);
          }
       });
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       getContentPane().setLayout(new BorderLayout());
       {
          panelDibujo = new PDibujo();
          getContentPane().add(
          panelDibujo, BorderLayout.CENTER);
       }
       setSize(400, 300);
   }
}
```

12.1 Clase Graphics

La clase *Graphics* permite realizar dibujos sobre el *JPanel* a través del método *Paint*. Esta clase está directamente relacionada con la clase *Color*, porque a través de esta se puede definir un color con el cual la instancia de *Graphics* puede pintar. La clase *Color* proporciona colores del formato *RGB*, los cuales pueden ser incluidos en el constructor.

Para graficar es importante tener en cuenta las coordenadas que se utilizan en computación. Estas coordenadas inician el (*0,0*) en la esquina superior izquierda y terminan en (*x,y*) en la esquina inferior derecha en donde *x,y* equivalen al ancho y alto del panel, respectivamente. La Figura 60 ilustra el manejo de coordenadas.

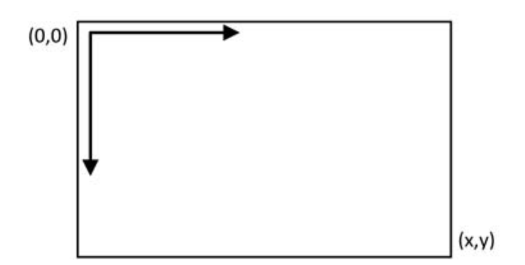


Figura 60. Coordenadas gráficas

12.1.1 Formas de *Graphics*

A través de la clase *Graphics* es posible dibujar una gran cantidad de formas como rectángulos, arcos, imágenes, textos, óvalos, polígonos y polilíneas.

Línea

Una línea se puede obtener mediante el siguiente método:

```
drawLine(int x1, int y1, int x2, int y2)
```

Este método dibuja una línea con un punto en la en la coordenada (*x1,y1*) y el otro punto en la coordenada (*x2,y2*).

Rectángulo

Un rectángulo se puede obtener mediante diferentes métodos como:

```
drawRect(int x, int y, int width, int height)
```

Este método dibuja un rectángulo con la esquina superior izquierda en la coordenada (*x,y*). Tiene un ancho y alto dados por los parámetros *width* y *height*. Esto indica que la coordenada de la esquina inferior derecha es (*x+width,y+height*).

```
drawRoundRect(int x, int y, int width, int height, int arcWidth,
int arcHeight)
```

Este método dibuja un rectángulo con las mismas características de posición y tamaño del método *drawRect*. Adicionalmente, dibuja las esquinas redondeadas con base en los parámetros *arcWidth* y *arcHeight*.

```
fillRect(int x, int y, int width, int height)
```

Este método dibuja un rectángulo con las mismas características de posición y tamaño del método *drawRect*. Adicionalmente, llena de color el rectángulo.

```
fillRoundRect(int x, int y, int width, int height, int arcWidth,
int arcHeight)
```

Este método dibuja un rectángulo con las mismas características de posición, tamaño y esquinas del método *drawRoundRect*. Adicionalmente, llena de color el rectángulo.

Óvalo

Un óvalo se puede obtener mediante diferentes métodos como:

```
drawOval(int x, int y, int width, int height)
```

Este método dibuja un óvalo con la esquina superior izquierda en la coordenada (*x,y*). Tiene un ancho y alto dados por los parámetros *width* y *heigh*. Esto indica que la coordenada de la esquina inferior derecha es (*x+width,y+height*).

```
fillOval(int x, int y, int width, int height)
```

Este método dibuja un óvalo con las mismas características de posición y tamaño del método *drawOval*. Adicionalmente, llena de color el óvalo.

Polígono

Un polígono se puede obtener mediante diferentes métodos como:

```
drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
```

Este método dibuja un polígono con puntos definidos por los arreglos *xPoints* y *yPoints*. La cantidad de puntos se incluyen en el parámetro *nPoints*.

```
fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
```

Este método dibuja un polígono con las mismas características de posición del método *drawPoligon*. Adicionalmente, llena de color el polígono.

Cadenas de caracteres

Una línea se puede obtener mediante el siguiente método:

```
drawString(String str, int x, int y)
```

Este método dibuja la cadena del parámetro *str* con la esquina inferior izquierda en la coordenada (*x, y*).

Imágenes

Una imagen se puede obtener mediante el siguiente método:

```
drawImage(Image img, int x, int y, ImageObserver observer)
```

Este método dibuja una imagen *img* con la esquina superior izquierda en la coordenada (*x,y*). El *ImageObserver* corresponde al contenedor que visualiza la imagen.

Considerando un *JPanel* se pueden realizar diferentes formas desde el método *Paint*, que recibe por parámetro un objeto *Graphics*. Este *JPanel* debe ser incluido en un *JFrame.*

Clase *PDibujo*

```
package graficas.formasBasicas;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;

import javax.swing.JPanel;

public class PDibujo extends JPanel {

   public void paint(Graphics g){
       Color c;
       c=new Color(255,255,255);
       g.setColor(c);
       g.fillRect(0, 0, getWidth(), getHeight());

       c=new Color(0,0,255);
       g.setColor(c);
       g.drawLine(10, 10, 10, 100);

       c=new Color(0,255,0);
       g.setColor(c);
       g.drawRect(20, 10, 50, 100);
       g.fillRect(80, 10, 50, 100);
       g.drawRoundRect(140, 10, 50, 100, 20, 20);
       g.fillRoundRect(200, 10, 50, 100, 20, 20);

       c=new Color(255,0,0);
       g.setColor(c);
       g.drawOval(260, 10, 50, 100);
       g.fillOval(320, 10, 50, 100);

       c=new Color(255,0,255);
       g.setColor(c);
       int []x={10,20,30,40,50};
       int []y={120,180,150,180,120};
       g.drawPolygon(x, y, 5);

       int []x2={60,70,80,90,100};
```

```
        int []y2={120,180,150,180,120};
        g.fillPolygon(x2, y2, 5);

        c=new Color(100,100,100);
        g.setColor(c);
        Font f = new Font("Tahoma",10,25);
        g.setFont(f);
        g.drawString("Hola mundo", 120, 150);

        Image image = Toolkit.getDefaultToolkit().getImage(
        "img/java.jpg");
        g.drawImage(image, 280, 120, this);
    }
}
```

Clase *FPrincipal*

```
package graficas.formasBasicas;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
    private PDibujo panelDibujo;

    public static void main(String[] args) {
        FPrincipal frame = new FPrincipal();
        frame.setVisible(true);
    }

    public FPrincipal() {
        initGUI();
        panelDibujo = new PDibujo();
        getContentPane().add(panelDibujo, BorderLayout.CENTER);
    }

    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Frame Principal");
        getContentPane().setLayout(new BorderLayout());
        setSize(400, 300);
    }
}
```

En esta aplicación dibuja las siguientes formas:

- Una línea con el primer punto en las coordenadas (*10,10*) y el segundo punto en las coordenadas (*10,100*).
- Un rectángulo sin relleno con el punto superior izquierdo en las coordenadas (*20,10*) y el punto inferior derecho en las coordenadas (*50,100*).
- Un rectángulo con relleno con el punto superior izquierdo en las coordenadas (*80,10*) y el punto inferior derecho en las coordenadas (*50,100*).
- Un rectángulo sin relleno con el punto superior izquierdo en las coordenadas (*140,10*) y el punto inferior derecho en las coordenadas (*50,100*). Además, tiene bordes redondeados con un alto y ancho de 20 píxeles, es decir, desde la esquina hasta donde inicia la curva de redondeo hay 20 píxeles tanto en el eje *X* como en *Y.*
- Un rectángulo con relleno con el punto superior izquierdo en las coordenadas (*200,10*) y el punto inferior derecho en las coordenadas (*50,100*). Además, tiene bordes redondeados con un alto y ancho de 20 píxeles, es decir, desde la esquina hasta donde inicia la curva de redondeo hay 20 píxeles tanto en el eje *X* como en *Y.*
- Un óvalo sin relleno con el punto superior izquierdo en las coordenadas (*260,10*) y el punto inferior derecho en las coordenadas (*50,100*). Esto indica que el óvalo tiene su punto máximo superior en la coordenada *Y=10*, su punto máximo a la izquierda en la coordenada *X=260*, su punto máximo inferior en la coordenada *Y=10+100=110* y su punto máximo a la derecha en la coordenada Y=260+50=310.
- Un óvalo con relleno con el punto superior izquierdo en las coordenadas (*320,10*) y el punto inferior derecho en las coordenadas (*50,100*).

- Un polígono con 5 puntos sin relleno.
- Un polígono con 5 puntos con relleno.
- La palabra “Hola mundo”, con fuente Tahoma y tamaño 25 píxeles en la coordenada inferior izquierda (*120,150*).
- Una imagen del logotipo de Java en la coordenada (*280,120*).

El resultado es como se presenta en la Figura 61.

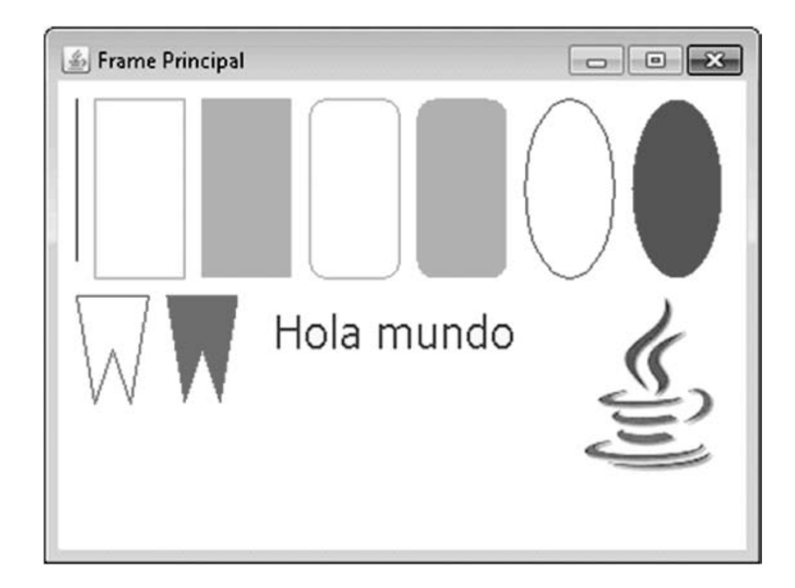


Figura 61. Formas de Graphics

12.1.2 Paneles estáticos y dinámicos

Si se desea pintar una cuadrícula de color gris colocando de fondo color blanco, se podría realizar la siguiente implementación en la clase *PDibujo* con base en que la resolución de este panel, es de 360 píxeles de ancho por 240 píxeles de alto como se definió en el método *setBounds*. Esta implementación utiliza el método *fillRect* para pintar un cuadrado blanco del tamaño del panel y el método *drawLine* para pintar cada línea en el panel.

Clase *PDibujo*

```
package graficas.panelEstatico;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class PDibujo extends JPanel {

   public void paint(Graphics g){
       Color c;
       c=new Color(255,255,255);
       g.setColor(c);
       g.fillRect(0, 0, 360, 240);
       c=new Color(180,180,180);
       g.setColor(c);
       int i;
       for(i=36; i<360; i+=36){
          g.drawLine(i, 0, i, 240);
       }
       for(i=24; i<240; i+=24){
              g.drawLine(0, i, 360, i);
       }
   }
}
```

Clase *FPrincipal*

```
package graficas.panelEstatico;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }
```

```
    public FPrincipal() {
        initGUI();
        panelDibujo = new PDibujo();
        panelDibujo.setBounds(10, 10, 360, 240);
        getContentPane().add(panelDibujo);
    }

    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Frame Principal");
        getContentPane().setLayout(null);
        setSize(400, 300);
    }
}
```

El resultado es como se muestra en la Figura 62.

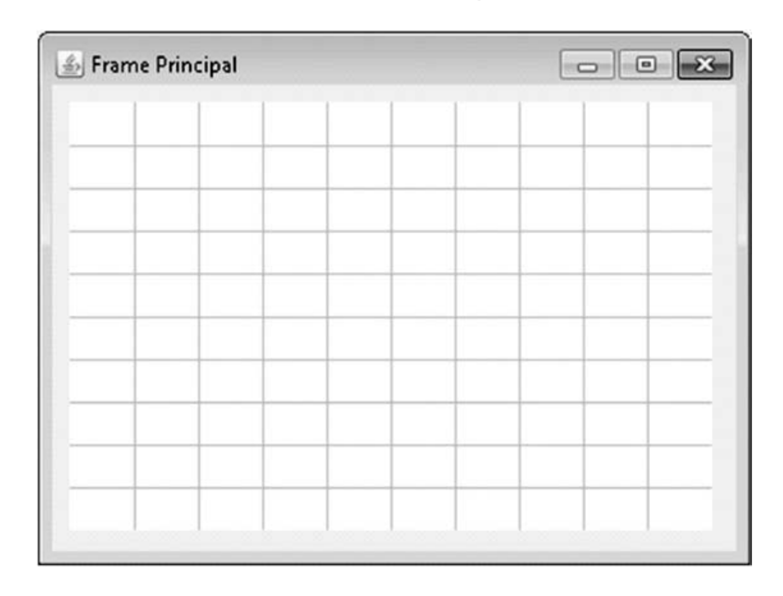


Figura 62. Gráfica con cuadrícula estática

En esta implementación, la cuadrícula es estática debido a que el tamaño del panel también lo es. El tamaño de cada celda entonces es de 36 x 24 píxeles para lograr una cuadrícula de 10 filas por 10 columnas. Esta implementación genera un defecto que consiste en

que al maximizar o cambiar de tamaño el *JFrame*, la gráfica no se adapta al nuevo tamaño.

Modificando el *Layout* del *JFrame* a *Border Layout* se obtiene un panel dinámico y su cuadrícula también lo sería. La implementación de las dos clases es la siguiente:

Clase *PDibujo*

```
package graficas.panelDinamico;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class PDibujo extends JPanel {
   private int ancho;
   private int alto;

   public void paint(Graphics g){
       this.ancho=(int)this.getSize().getWidth();
       this.alto=(int)this.getSize().getHeight();
       int anchoCelda=(int)(this.getSize().getWidth()/10);
       int altoCelda=(int)(this.getSize().getHeight()/10);
       Color c;
       c=new Color(255,255,255);
       g.setColor(c);
       g.fillRect(0, 0, this.ancho, this.alto);
       c=new Color(180,180,180);
       g.setColor(c);
       int i;
       for(i=anchoCelda; i<this.ancho-anchoCelda; i+
       =anchoCelda){
          g.drawLine(i, 0, i, this.alto);
       }
       for(i=altoCelda; i<this.alto-altoCelda; i+=altoCelda){
          g.drawLine(0, i, this.ancho, i);
       }
   }
}
```

Clase *FPrincipal*

```
package graficas.panelDinamico;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
       panelDibujo = new PDibujo();
       getContentPane().add(panelDibujo, BorderLayout.CENTER);

   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       getContentPane().setLayout(new BorderLayout());
       setSize(400, 300);
   }
}
```

En este caso se debe calcular el ancho y alto del panel a través de los *métodosgetSize().getWidth()* y *getSize().getHeight()*. Así mismo, el alto y ancho de cada celda, el cual corresponde al ancho y alto del panel dividido en 10, ya que ese es el número de celdas deseadas. Se debe hacer *casting* a *int*, debido a que el método *drawLine* recibe en sus parámetros este tipo de dato. De esta forma, al cambiar el tamaño del *Frame*, automáticamente cambia el tamaño de la cuadrícula.

El resultado es como se muestra en la Figura 63.

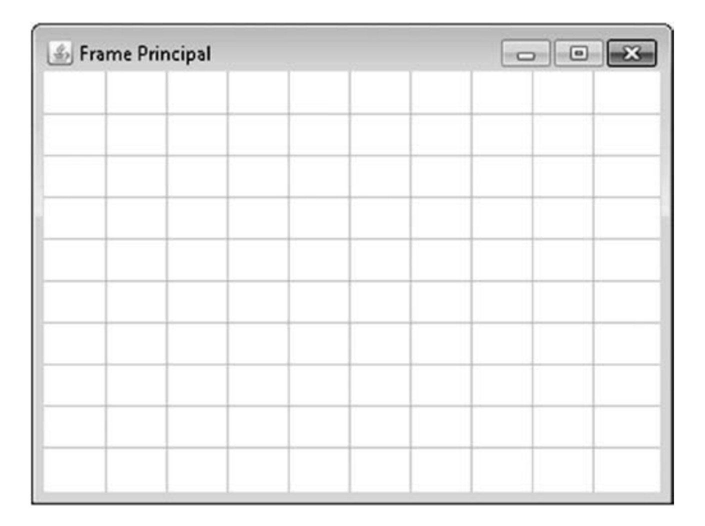


Figura 63. Gráfica con cuadrícula dinámica

12.2 Gráficas de señales

Para graficar señales como seno y coseno, lo más apropiado es crear una capa de lógica de negocio que contenga las clases seno y coseno. Sin embargo, para asegurar escalabilidad al proyecto se debería crear una clase abstracta llamada "*Senal*" que herede atributos y métodos necesarios para el dibujo. La implementación es la siguiente:

Clase *Senal*

```
package graficas.senales.logica;

public abstract class Senal {
   protected int amplitud;
   protected int frecuencia;
   protected int offset;

   public Senal(int a, int f,int o){
       this.amplitud=a;
       this.frecuencia=f;
       this.offset=o;
   }

   public int getAmplitud() {
       return amplitud;
```

```
    }

    public void setAmplitud(int amplitud) {
        this.amplitud = amplitud;
    }

    public int getFrecuencia() {
        return frecuencia;
    }

    public void setFrecuencia(int frecuencia) {
        this.frecuencia = frecuencia;
    }

    public int getOffset() {
        return offset;
    }

    public void setOffset(int offset) {
        this.offset = offset;
    }

    public abstract int[] calcular(int ancho, int alto);
}
```

Clase *Seno*

```
package graficas.senales.logica;

public class Seno extends Senal {

    public Seno(int a, int f,int o){
        super(a,f,o);
    }

    @Override
    public int[] calcular(int ancho, int alto){
        int [] puntos = new int[ancho];
        for(int i=0;i<ancho;i++){
           puntos[i]=(this.offset*alto/10)+(int)((alto/10)
           *this.amplitud* Math.sin(Math.PI/180*i
           *this.frecuencia));
        }
        return puntos;
    }
}
```

Clase *Coseno*

```
package graficas.senales.logica;

public class Coseno extends Senal {

   public Coseno(int a, int f,int o){
       super(a,f,o);
   }

   @Override
   public int[] calcular(int ancho, int alto){
       int [] puntos = new int[ancho];
       for(int i=0;i<ancho;i++){
          puntos[i]=(this.offset*alto/10)+(int)((alto/10)
          *this.amplitud* Math.cos(Math.PI/180*i*this.frecuencia));
       }
       return puntos;
   }
}
```

Las clases *Seno* y *Coseno* implementan un método llamado *calcular* que retorna un arreglo de puntos. Estos puntos son los valores de la señal con la amplitud, frecuencia y *offset* (desfase en el eje *Y*) asignados a través del constructor. Los parámetros ancho y alto indican la cantidad de píxeles del panel, debido a que este es dinámico.

Los puntos de la señal se calculan haciendo un proceso iterativo que depende del número de píxeles en el eje *X*. Los puntos dependen del nivel de *offset* que se multiplican por el alto del panel y divide en 10 debido a que la cuadrícula posee 10 filas. Al nivel *offset* se le suma la señal, cuya amplitud se multiplica por el atributo correspondiente por el alto dividido en 10, con el fin que la señal ocupe en el eje *Y* una amplitud proporcional a las filas de la cuadrícula. Las señales seno y coseno reciben por parámetro el ángulo en radianes, por lo que se debe realizar la conversión a frecuencia multiplicando por *pi* y dividiendo por 180.

El *JFrame*, entonces, debe cambiar para ofrecer un mecanismo que asigne valores a las señales. En este caso se proponen tres cuadros de texto para los valores, un botón para dibujar la señal seno y un botón para dibujar la señal coseno. Para estos controles, lo más apropiado

es crear una clase que extienda de *JPanel*. Así mismo, se requiere una clase para dibujar que extienda de *JPanel* que contenga el método *Paint*. El *frame* debe tener objetos que sean instancias de estas clases, de esta forma, para que el panel de controles acceda al panel de dibujo es necesario que el panel de controles tenga una referencia del *frame*.

La implementación es la siguiente:

Clase *FPrincipal*

```
package graficas.senales.presentacion;

import graficas.senales.logica.Coseno;
import graficas.senales.logica.Senal;
import graficas.senales.logica.Seno;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JLabel labelTitulo;
   private PDibujo panelDibujo;
   private PControles panelControles;
   private Senal senal;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       getContentPane().setLayout(new BorderLayout());
       this.setTitle("Senales Trigonometricas");
       {
          labelTitulo = new JLabel();
          getContentPane().add(labelTitulo, BorderLayout.NORTH);
```

```
            labelTitulo.setText("SENALES TRIGONOMETRICAS");
            labelTitulo.setFont(new java.awt.Font("Tahoma",1,16));
            labelTitulo.setHorizontalAlignment(
            SwingConstants.CENTER);
         }
         {
            panelDibujo = new PDibujo();
            getContentPane().add(panelDibujo,
            BorderLayout.CENTER);
         }
         {
            panelControles = new PControles(this);
            getContentPane().add(panelControles,
            BorderLayout.SOUTH);
         }
         setSize(400, 300);
      }

      public void pintarSeno(int amplitud, int frecuencia,
      int offset){
          senal=new Seno(amplitud, frecuencia, offset);
          panelDibujo.actualizar(senal);
          }

      public void pintarCoseno(int amplitud, int frecuencia,
      int offset){
          senal=new Coseno(amplitud, frecuencia, offset);
          panelDibujo.actualizar(senal);
      }
}
```

Clase *PControles*

```
package graficas.senales.presentacion;

import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class PControles extends JPanel {
   private FPrincipal framePrincipal;
   private JLabel labelAmplitud;
   private JTextField textAmplitud;
   private JLabel labelFrecuencia;
```

```
private JTextField textFrecuencia;
private JLabel labelOffset;
private JTextField textOffset;
private JButton buttonSeno;
private JButton buttonCoseno;

public PControles(FPrincipal framePricipal) {
    initGUI();
    this.framePrincipal = framePricipal;
}

private void initGUI() {
    {
       labelAmplitud = new JLabel();
       add(labelAmplitud);
       labelAmplitud.setText("Amplitud");
    }
    {
       textAmplitud = new JTextField();
       add(textAmplitud);
       textAmplitud.setText("1");
       textAmplitud.setPreferredSize(new Dimension(20, 20));
    }
    {
       labelFrecuencia = new JLabel();
       add(labelFrecuencia);
       labelFrecuencia.setText("Frecuencia");
    }
    {
       textFrecuencia = new JTextField();
       add(textFrecuencia);
       textFrecuencia.setText("10");
       textFrecuencia.setPreferredSize(new Dimension(20, 20));
    }
    {
       labelOffset = new JLabel();
       add(labelOffset);
       labelOffset.setText("Offset");
    }
    {
       textOffset = new JTextField();
       add(textOffset);
       textOffset.setText("0");
       textOffset.setPreferredSize(new Dimension(20, 20));
    }
    {
       buttonSeno = new JButton();
       add(buttonSeno);
       buttonSeno.setText("Seno");
       buttonSeno.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent evt) {
```

```
                buttonSenoActionPerformed(evt);
            }
        });
    }
    {
        buttonCoseno = new JButton();
        add(buttonCoseno);
        buttonCoseno.setText("Coseno");
        buttonCoseno.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                buttonCosenoActionPerformed(evt);
            }
        });
    }
}

private void buttonSenoActionPerformed(ActionEvent evt) {
    int amplitud = Integer.parseInt(textAmplitud.getText());
    int frecuencia = Integer.parseInt(
    textFrecuencia.getText());
    int offset = Integer.parseInt(textOffset.getText());
    framePrincipal.pintarSeno(amplitud, frecuencia, offset);
}

private void buttonCosenoActionPerformed(ActionEvent evt) {
    int amplitud = Integer.parseInt(textAmplitud.getText());
    int frecuencia = Integer.parseInt(
    textFrecuencia.getText());
    int offset = Integer.parseInt(textOffset.getText());
    framePrincipal.pintarCoseno(amplitud, frecuencia,
    offset);
}
}
```

Clase *PDibujo*

```
package graficas.senales.presentacion;

import graficas.senales.logica.Senal;

import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JPanel;

public class PDibujo extends JPanel {
    private int ancho;
    private int alto;
    private Senal senal;
```

```
    public void actualizar(Senal senal){
        this.senal=senal;
        this.repaint();
    }

    public void paint(Graphics g){
        this.ancho=(int)this.getSize().getWidth();
        this.alto=(int)this.getSize().getHeight();
        int anchoCelda=(int)(this.getSize().getWidth()/10);
        int altoCelda=(int)(this.getSize().getHeight()/10);
        Color c;
        c=new Color(255,255,255);
        g.setColor(c);
        g.fillRect(0, 0, this.ancho, this.alto);
        c=new Color(180,180,180);
        g.setColor(c);
        int i;
        for(i=anchoCelda; i<this.ancho-(anchoCelda/2); i+
        =anchoCelda){
           g.drawLine(i, 0, i, this.alto);
        }
        for(i=altoCelda; i<this.alto-(altoCelda/2); i+=altoCelda){
           g.drawLine(0, i, this.ancho, i);
        }
        c=new Color(0,0,255);
        g.setColor(c);
        if(this.senal!=null){
           int []puntos = this.senal.calcular(this.ancho,this.alto);
           for(i=0;i<this.ancho-1; i++){
               g.drawLine(i, (this.alto/2)-puntos[i], i+1, (
               this.alto/2)-puntos[i+1]);
           }
        }

    }
}
```

Las anteriores clases presentan las siguientes características:

- El *frame* tiene un panel en el centro con la gráfica.
- El *frame* tiene un panel en el sur con los componentes necesarios para enviar los parámetros de amplitud, frecuencia y *offset* de las señales.
- El panel *PControles* posee un método para el botón *Seno* que hace un llamado al metido *pintarSeno* que crea un objeto instancia de la clase *Seno* en la capa de lógica, al que se le envía por el constructor los valores de amplitud, frecuencia y *offset*.

Este método envía el objeto al objeto *panelDibujo* a través del método actualizar.

- El panel *PControles* posee un método para el botón *Coseno* que hace un llamado al metido *pintarCoseno* que crea un objeto instancia de la clase *Coseno* en la capa de lógica, al que se le envía por el constructor los valores de amplitud, frecuencia y *offset*. Este método envía el objeto al objeto *panelDibujo* a través del método actualizar.
- El panel *PDibujo* tiene un método actualizar que asigna el objeto *senal.*
- El panel *PDibujo* tiene un método *Paint* que coloca el fondo blanco al panel, la cuadrícula y dibuja la *senal* basado en los puntos retornados por *Seno* y *Coseno*. Pintar la gráfica consiste en pintar un conjunto de líneas desde un punto hasta el punto siguiente. El método *drawLine* permite dibujar las líneas en mención. Los parámetros del eje *Y*, tienen un valor que depende del alto del panel, con el fin de ubicar la *senal* en la mitad.

Los resultados se presentan en las figuras 64 y 65.

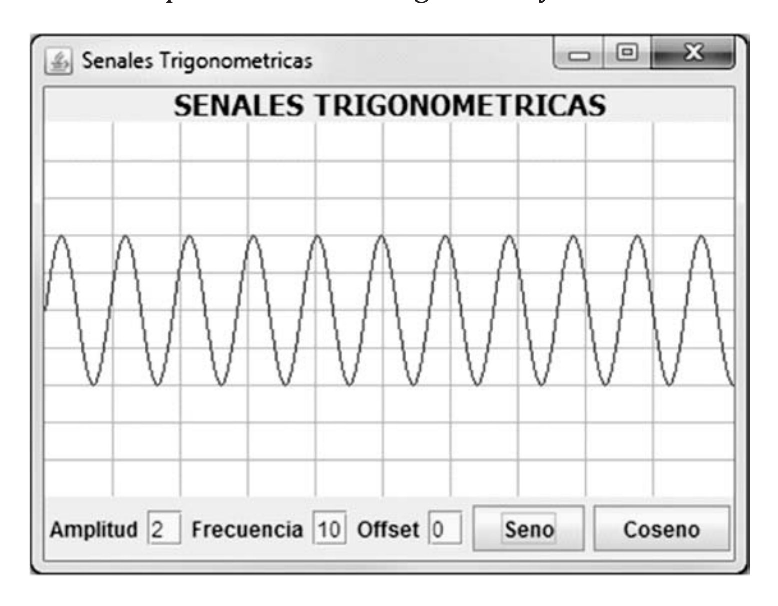


Figura 64. Dibujo de la senal Seno

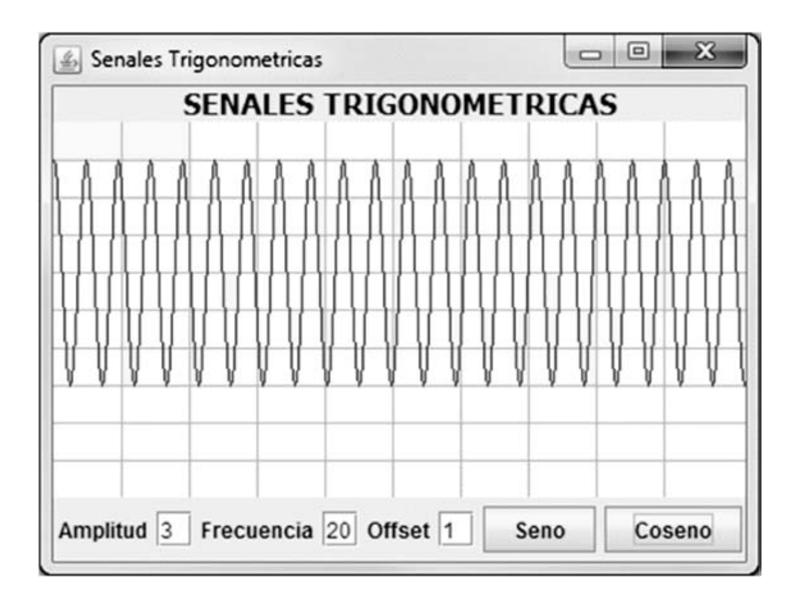


Figura 65. Dibujo de la senal Coseno

12.3 Clase *Graphics2D*

La clase *Graphics2D* permite una gran cantidad de funcionalidades sobre las formas que se pintan en un *JPanel*. Entre estas funcionalidades se encuentran degradado, transparencia y rotación, entre otros. *Graphics2D* es una subclase de *Graphics* lo que indica que un objeto *Graphics* puede ser convertido a un objeto *Graphics2D*. De esta forma, cuando se requiera utilizar un objeto de *Graphics2D* en un *JPanel*, se debe crear un objeto *Graphics2D* al cual se inicializa mediante un *casting* del objeto *Graphics*. La sintaxis es la siguiente:

```
public void paint(Graphics g){
   Graphics2D g2d = (Graphics2D)g;
}
```

12.3.1 Degradado

Para lograr un degradado es necesario utilizar el método *setPaint* de la clase *Graphics2D*. Este método debe recibir por parámetro un

objeto de la clase *GradientPaint* el cual contiene la configuración del degradado que se quiere asignar a las formas pintadas mediante *Graphics2D*. Por ejemplo, si se desea pintar una elipse de color negra con degradado diagonal, la implementación es la siguiente:

Clase *FPrincipal*

```
package graficas2D.degradado;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
       panelDibujo = new PDibujo();
       getContentPane().add(panelDibujo, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       getContentPane().setLayout(new BorderLayout());
       setSize(400, 300);
   }
}
```

Clase *PDibujo*

```
package graficas2D.degradado;

import java.awt.Color;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Shape;
import java.awt.geom.Ellipse2D;

import javax.swing.JPanel;
```

```
public class PDibujo extends JPanel {

   public void paint(Graphics g){
       Graphics2D g2d = (Graphics2D)g;
       GradientPaint gradient = new GradientPaint(0, 0,
       Color.black, getWidth(), getHeight(), Color.white, true);
       g2d.setPaint(gradient);
       g2d.fillOval(0,0,getWidth(),getHeight());
   }
}
```

El resultado se observa en la Figura 66.

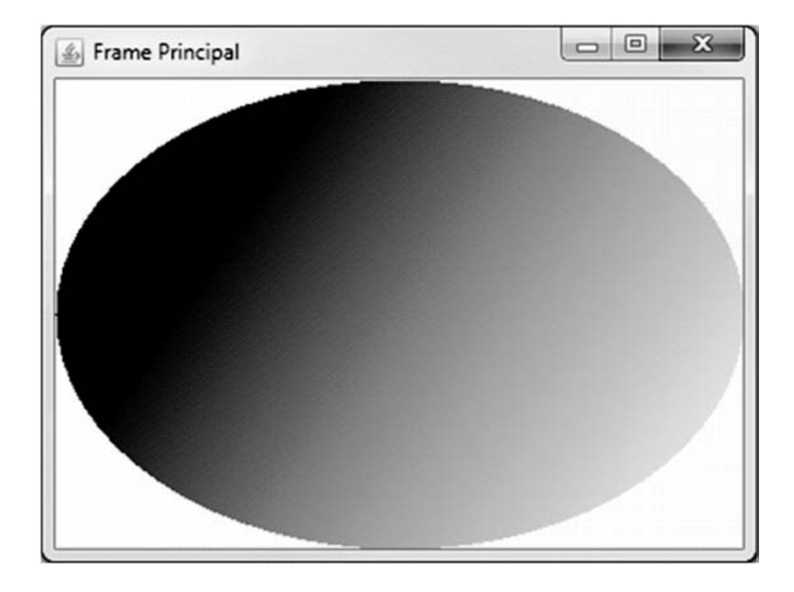


Figura 66. Degradado con Graphics2D

12.3.2 Transparencia

Para lograr transparencia es necesario utilizar el método *setComposite* de la clase *Graphics2D*. Este método debe recibir por parámetro un objeto de la clase *Composite*. Este objeto se crea mediante la clase *AlphaComposite* en el método *getInstance* con el que se lleva a cabo la configuración para la transparencia. Por ejemplo, si se desea pintar dos cuadrados superpuestos con transparencia, la implementación es la siguiente:

Clase *FPrincipal*

```
package graficas2D.transparencia;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
       panelDibujo = new PDibujo();
       getContentPane().add(panelDibujo, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       getContentPane().setLayout(new BorderLayout());
       setSize(400, 300);
   }
}
```

Clase *PDibujo*

```
package graficas2D.transparencia;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }
```

```
    public FPrincipal() {
        initGUI();
        panelDibujo = new PDibujo();
        getContentPane().add(panelDibujo, BorderLayout.CENTER);
    }

    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Frame Principal");
        getContentPane().setLayout(new BorderLayout());
        setSize(400, 300);
    }
}
```

El resultado es el que aparece en la Figura 67.

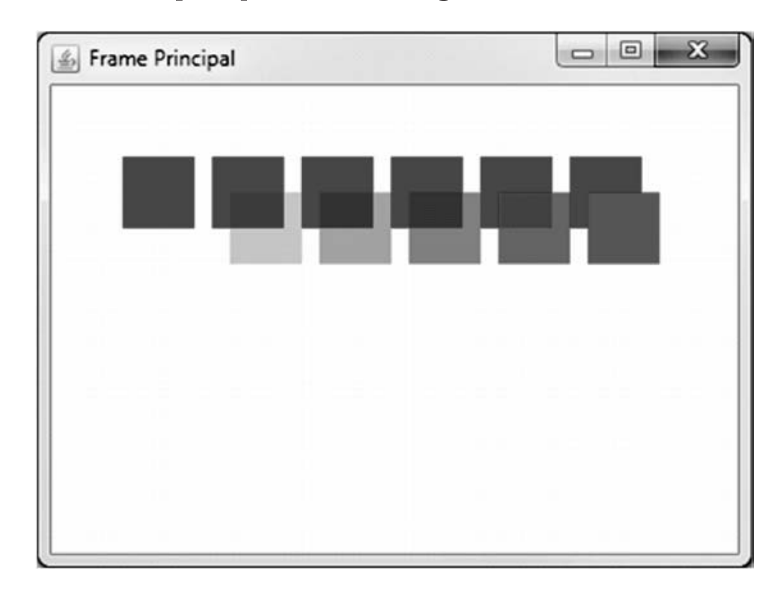


Figura 67. Transparencia con Graphics2D

12.3.3 Translación y rotación

Para lograr translación es necesario utilizar el método "*translate*" de la clase "*Graphics2D*". Este método debe recibir por parámetro las coordenadas de translación. Estas coordenadas equivalen al nuevo centro del objeto *Graphics2D*. Para lograr rotación es necesario utilizar el método "*rotate*" de la clase "*Graphics2D*". Este método debe recibir

por parámetro el ángulo en radianes de rotación. Por ejemplo, si se desea pintar una palabra que rote sobre su eje, la implementación es la siguiente:

Clase FPrincipal

```
package graficas2D.rotacionTranslacion;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private PDibujo panelDibujo;

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   public FPrincipal() {
       initGUI();
       panelDibujo = new PDibujo();
       getContentPane().add(panelDibujo, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Frame Principal");
       getContentPane().setLayout(new BorderLayout());
       setSize(400, 300);
   }
}
```

Clase PDibujo

```
package graficas2D.rotacionTranslacion;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JPanel;
```

```
public class PDibujo extends JPanel {

   public void paint(Graphics g){
       Graphics2D g2d = (Graphics2D)g;
       g2d.translate(getWidth()/2, getHeight()/2);
       g2d.setFont(new Font("Times",10,50));
       for(int i=0; i<8; i++){
       g2d.setColor((i%2==0)?Color.BLACK:Color.BLUE);
       g2d.rotate(Math.PI/4);
       g2d.drawString("Java", 0, 0);
       }
   }
}
```

El resultado es el siguiente:

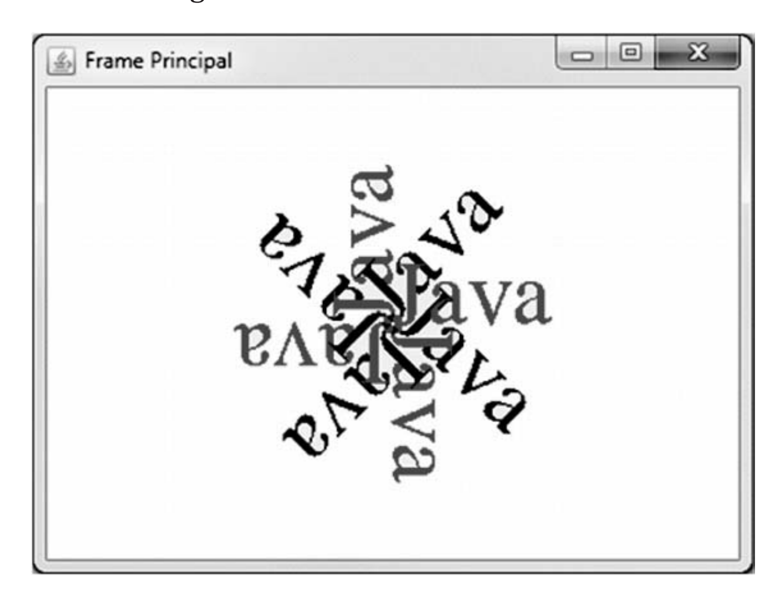


Figura 68. Translación y rotación con Graphics2D

12.4 Gráficas estadísticas (Chart)

Existen componentes desarrolladores por diferentes colaboradores de Java para la construcción de gráficas estadísticas. Uno de los

componentes más conocidos es *JFreeChart*[1]. Este componente permite de forma simple la creación de diferentes tipos de gráficos con base en un conjunto de datos que se proporcionan a través de métodos específicos del componente.

Para usar *JFreeChart* es necesario descargar el componente del sitio *web* e incluir en el proyecto como librería al menos los elementos presentados en la Figura 69.

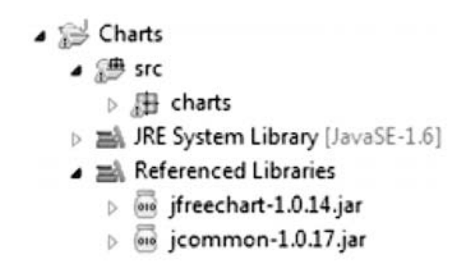


Figura 69. *Referenced Libraries* para *JFreeChart*

12.4.1 Diagramas de torta

JFreeChart permite la creación de diagrama de torta (*Pie Chart*) en dos o tres dimensiones. Además, también permite almacenar el resultado en un archivo *jpg*.

En el momento de crear un *Pie Chart* es necesario asignar los datos mediante la clase *DefaultPieDataSet* en el método *setValue*. Este método permite incluir un texto y un valor numérico por cada valor que contendrá el *Pie Chart*. Sin embargo, para pintar el *Pie Chart* es necesario realizar un *casting* a la clase *PieDataSet*. La clase *PieDataSet* es súper clase de *DefaultPieDataSet*. La sintaxis es la siguiente:

```
DefaultPieDataset defaultDataSet = new DefaultPieDataset();
defaultDataSet.setValue("Texto 1", 40);
defaultDataSet.setValue("Texto 2", 60);
PieDataset dataSet = defaultDataSet;
```

[1] http://www.jfree.org/jfreechart/

Una vez asignados los datos se crea un *chart* mediante la clase *ChartFactory*. El *chart* debe tener un título, datos y puede tener leyenda y texto flotante. La sintaxis es la siguiente:

```
JFreeChart chart = ChartFactory.createPieChart("Pie", dataSet,
true, true, true);

Posteriormente, el Pie Chart debe desplegarse mediante un
ChartPanel. La sintaxis es la siguiente:

ChartPanel panel = new ChartPanel(chart);
```

JFreeChart también permite exportar el resultado como una imagen *jpg*. Para ello se debe tener un objeto de la clase *File* que indique el archivo en donde se almacenara la imagen, el *chart* y el tamaño en píxeles. La sintaxis es la siguiente:

```
File file = new File("./img/pie.jpg");
try {
   ChartUtilities.saveChartAsJPEG(file, chart, 800, 600);
} catch (IOException e) {
   e.printStackTrace();
}
```

Ejemplo de *Pie Chart 2D*

El siguiente código presenta un ejemplo de creación de un *Pie Chart 2D* con datos ingresados directamente en el código. La implementación es la siguiente:

```
package charts.pie;

import java.awt.BorderLayout;
import java.io.File;
import java.io.IOException;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;
```

```
import org.jfree.data.general.PieDataset;

public class FPie2D extends JFrame {

   public static void main(String[] args) {
       FPie2D frame = new FPie2D();
       frame.setVisible(true);
   }

   public FPie2D() {
       initGUI();
       PieDataset dataSet = new DefaultPieDataset();
       dataSet = createDataSet();
       JFreeChart chart = ChartFactory.createPieChart(
       "Uso de exploradores", dataSet, true, true, true);
       ChartPanel panel = new ChartPanel(chart);
       File file = new File("./img/pie.jpg");
       try {
          ChartUtilities.saveChartAsJPEG(file, chart, 800, 600);
       } catch (IOException e) {
          e.printStackTrace();
       }
       getContentPane().add(panel, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setLayout(new BorderLayout());
       setTitle("JFreePie Chart 2D");
       setSize(400, 300);
   }

   private PieDataset createDataSet() {
       DefaultPieDataset defaultDataSet =
       new DefaultPieDataset();
       defaultDataSet.setValue("Mozilla Firefox", 40);
       defaultDataSet.setValue("Google Chrome", 25);
       defaultDataSet.setValue("Internet Explorer", 22);
       defaultDataSet.setValue("Safari", 8);
       defaultDataSet.setValue("Opera", 5);
       return defaultDataSet;
   }
}
```

Los resultados son los siguientes:



Figura 70. *Pie Chart* 2D



Figura 71. *Pie Chart* 2D exportado como jpg

También se puede crear un *Pie Chart* en 3D. Este *Pie Chart 3D* permite además, especificar un ángulo y dirección. La implementación de un *Pie Chart 3D* con los mismos datos del ejemplo anterior es la siguiente:

```
package charts.pie;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PiePlot3D;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;
import org.jfree.util.Rotation;

public class FPie3D extends JFrame {

   public static void main(String[] args) {
       FPie3D frame = new FPie3D();
       frame.setVisible(true);
   }

   public FPie3D() {
       initGUI();
       PieDataset dataSet = new DefaultPieDataset();
       dataSet = createDataSet();
       JFreeChart chart = ChartFactory.createPieChart3D(
       "Uso de exploradores", dataSet, true, true, true);
       PiePlot3D plot = (PiePlot3D)chart.getPlot();
       plot.setStartAngle(0);
       plot.setDirection(Rotation.ANTICLOCKWISE);
       ChartPanel panel = new ChartPanel(chart);
       getContentPane().add(panel, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setLayout(new BorderLayout());
       setTitle("JFreePie Chart 2D");
       setSize(400, 300);
   }
```

```
    private PieDataset createDataSet() {
        DefaultPieDataset defaultDataSet =
        new DefaultPieDataset();
        defaultDataSet.setValue("Mozilla Firefox", 40);
        defaultDataSet.setValue("Google Chrome", 25);
        defaultDataSet.setValue("Internet Explorer", 22);
        defaultDataSet.setValue("Safari", 8);
        defaultDataSet.setValue("Opera", 5);
        return defaultDataSet;
    }
}
```

El resultado es el siguiente:

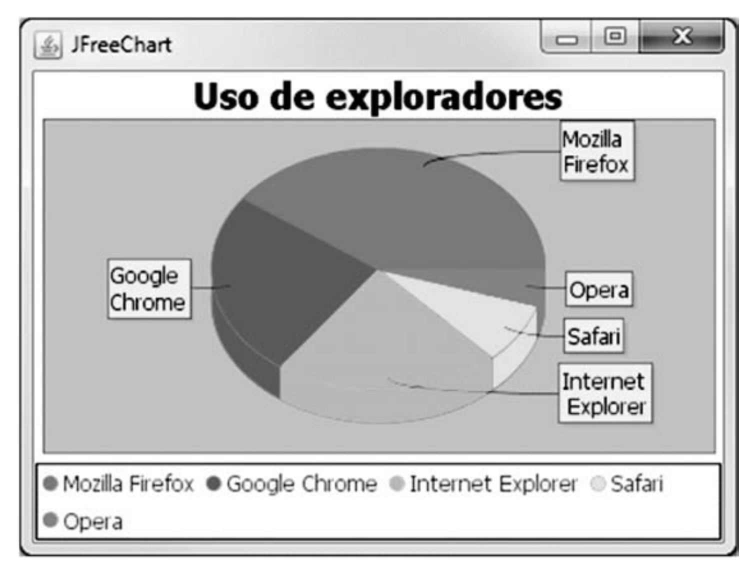


Figura 72. *Pie Chart* 3D

12.4.2 Diagramas de líneas, área y barras

JFreeChart permite la creación de diagrama de líneas (*Line Chart*), diagrama de área (*Area Chart*)y diagrama de barras (*Bar Chart*) en dos o tres dimensiones. En el momento de crear uno de estos *chart* es necesario asignar los datos mediante la clase *DefaultCategoryDataSet* en el método *setValue*. Este método permite incluir un valor que hará referencia a la elevación de la barra en el eje Y, un texto que indica una categoría y un texto que agrupa diferentes valores de categorías a un elemento que se visualiza en el eje X.

Ejemplo de *Line Chart 2D*

```
package charts.line;

import java.awt.BorderLayout;
import java.io.File;
import java.io.IOException;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;

public class FLine2D extends JFrame {

   public static void main(String[] args) {
       FLine2D frame = new FLine2D();
       frame.setVisible(true);
   }

   public FLine2D() {
       initGUI();
       CategoryDataset dataSet = new DefaultCategoryDataset();
       dataSet = createDataSet();
       JFreeChart chart = ChartFactory.createLineChart(
       "Calificaciones", "Estudiantes", "Calificacion", dataSet,
       PlotOrientation.VERTICAL, true, true, false);
       ChartPanel panel = new ChartPanel(chart);
       getContentPane().add(panel, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setLayout(new BorderLayout());
       setTitle("JFreeBar Chart 3D");
       setSize(400, 300);
   }

   private CategoryDataset createDataSet() {
       DefaultCategoryDataset defaultDataSet =
       new DefaultCategoryDataset();
```

```
        defaultDataSet.setValue(4.5, "POO", "Est1");
        defaultDataSet.setValue(4.2, "Calculo", "Est1");
        defaultDataSet.setValue(3.0, "Fisica", "Est1");
        defaultDataSet.setValue(5.0, "POO", "Est2");
        defaultDataSet.setValue(3.5, "Calculo", "Est2");
        defaultDataSet.setValue(3.9, "Fisica", "Est2");
        defaultDataSet.setValue(2.0, "POO", "Est3");
        defaultDataSet.setValue(3.6, "Calculo", "Est3");
        defaultDataSet.setValue(4.8, "Fisica", "Est3");
        defaultDataSet.setValue(3.1, "POO", "Est4");
        defaultDataSet.setValue(2.5, "Calculo", "Est4");
        defaultDataSet.setValue(3.8, "Fisica", "Est4");
        return defaultDataSet;
    }
}
```

El resultado es el siguiente:

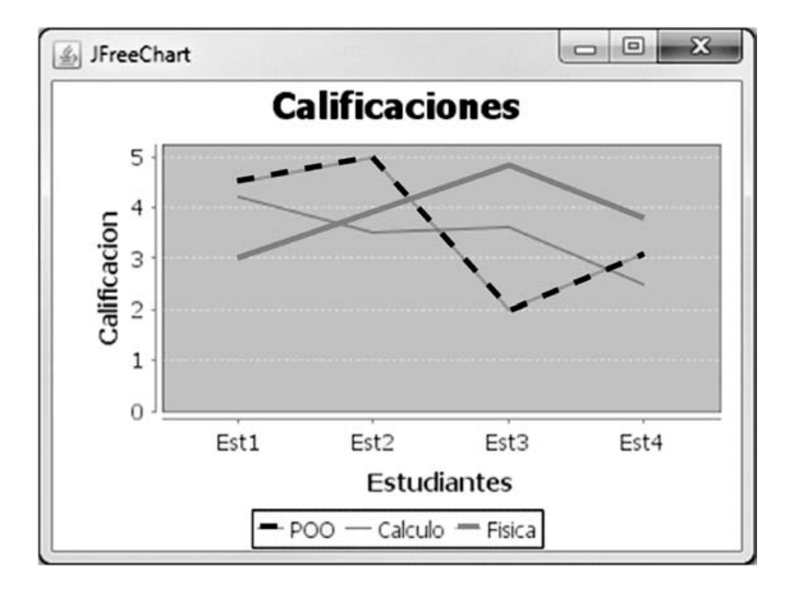


Figura 73. *Line Chart 2D*

Una aplicación interesante del *Char Line* y *Bar Chart* es que se puede ampliar un área especificada por el usuario, seleccionando con el *mouse* dicha área. En ese caso, si por ejemplo se selecciona el área entre 3 y 5 en el eje de las Y, el nuevo resultado es el siguiente:

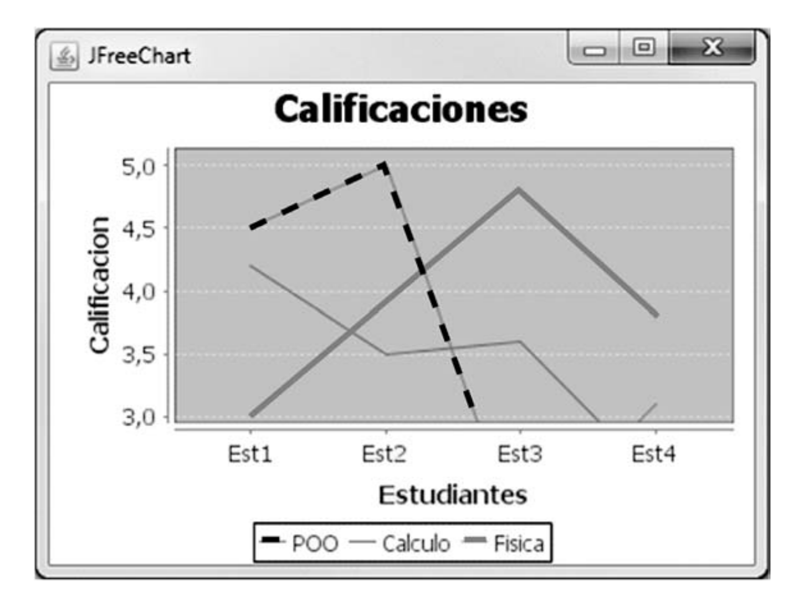


Figura 74. *Line Chart* 2D Ampliado

Ejemplo de *Area Chart*

El siguiente código presenta un ejemplo de creación de un *Area Chart2D* con datos ingresados directamente en el código. En este *chart* se puede aplicar transparencias mediante el método *setForegroundAlpha*. La implementación es la siguiente:

```
package charts.area;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;

public class FArea2D extends JFrame {
```

```
    public static void main(String[] args) {
        FArea2D frame = new FArea2D();
        frame.setVisible(true);
    }

    public FArea2D() {
        initGUI();
        CategoryDataset dataSet = new DefaultCategoryDataset();
        dataSet = createDataSet();
        JFreeChart chart = ChartFactory.createAreaChart(
        "Calificaciones", "Estudiantes", "Calificacion", dataSet,
        PlotOrientation.VERTICAL, true, true, false);
        chart.getCategoryPlot().setForegroundAlpha(0.5f);
        ChartPanel panel = new ChartPanel(chart);
        getContentPane().add(panel, BorderLayout.CENTER);
    }

    private void initGUI() {
        setDefaultCloseOperation(
        WindowConstants.DISPOSE_ON_CLOSE);
        setLayout(new BorderLayout());
        setTitle("JFreeChart Bar 3D");
        setSize(400, 300);
    }

    private CategoryDataset createDataSet() {
        DefaultCategoryDataset defaultDataSet =
        new DefaultCategoryDataset();
        defaultDataSet.setValue(4.5, "POO", "Est1");
        defaultDataSet.setValue(4.2, "Calculo", "Est1");
        defaultDataSet.setValue(3.0, "Fisica", "Est1");
        defaultDataSet.setValue(5.0, "POO", "Est2");
        defaultDataSet.setValue(3.5, "Calculo", "Est2");
        defaultDataSet.setValue(3.9, "Fisica", "Est2");
        defaultDataSet.setValue(2.0, "POO", "Est3");
        defaultDataSet.setValue(3.6, "Calculo", "Est3");
        defaultDataSet.setValue(4.8, "Fisica", "Est3");
        defaultDataSet.setValue(3.1, "POO", "Est4");
        defaultDataSet.setValue(2.5, "Calculo", "Est4");
        defaultDataSet.setValue(3.8, "Fisica", "Est4");
        return defaultDataSet;
    }

}
```

El resultado es el siguiente:

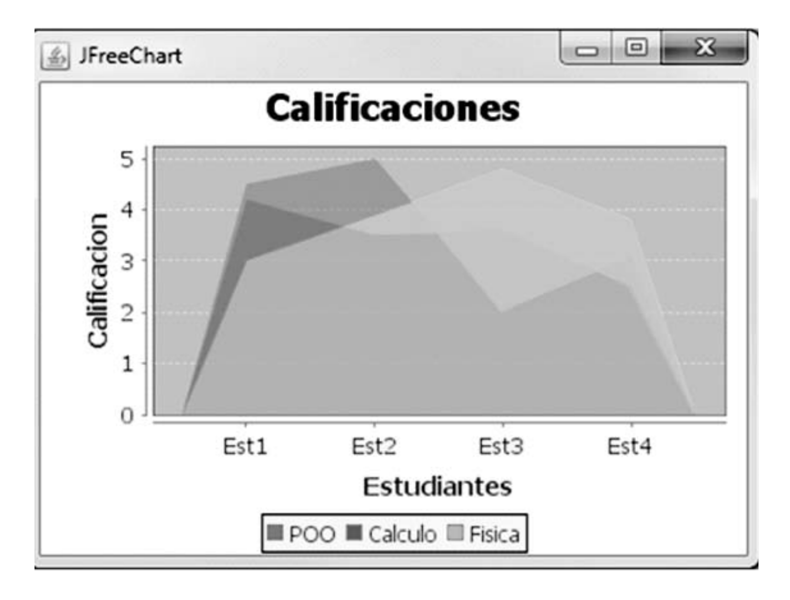


Figura 75. *Area Chart* 2D

Ejemplo de *Bar Chart 3D*

El siguiente código presenta un ejemplo de creación de un *Bar Chart 3D* con datos ingresados directamente en el código. La implementación es la siguiente:

```
package charts.bar;

import java.awt.BorderLayout;
import java.io.File;
import java.io.IOException;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
```

```
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;

public class FBar3D extends JFrame {

   public static void main(String[] args) {
       FBar3D frame = new FBar3D();
       frame.setVisible(true);
   }

   public FBar3D() {
       initGUI();
       CategoryDataset dataSet = new DefaultCategoryDataset();
       dataSet = createDataSet();
       JFreeChart chart = ChartFactory.createBarChart3D(
       "Calificaciones", "Estudiantes", "Calificacion", dataSet,
       PlotOrientation.VERTICAL, true, true, false);
       ChartPanel panel = new ChartPanel(chart);
       getContentPane().add(panel, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setLayout(new BorderLayout());
       setTitle("JFreeBar Chart 3D");
       setSize(400, 300);
   }

   private CategoryDataset createDataSet() {
       DefaultCategoryDataset defaultDataSet =
       new DefaultCategoryDataset();
       defaultDataSet.setValue(4.5, "POO", "Est1");
       defaultDataSet.setValue(4.2, "Calculo", "Est1");
       defaultDataSet.setValue(3.0, "Fisica", "Est1");
       defaultDataSet.setValue(5.0, "POO", "Est2");
       defaultDataSet.setValue(3.5, "Calculo", "Est2");
       defaultDataSet.setValue(3.9, "Fisica", "Est2");
       defaultDataSet.setValue(2.0, "POO", "Est3");
       defaultDataSet.setValue(3.6, "Calculo", "Est3");
       defaultDataSet.setValue(4.8, "Fisica", "Est3");
       defaultDataSet.setValue(3.1, "POO", "Est4");
       defaultDataSet.setValue(2.5, "Calculo", "Est4");
       defaultDataSet.setValue(3.8, "Fisica", "Est4");
       return defaultDataSet;
   }
}
```

El resultado es el siguiente:

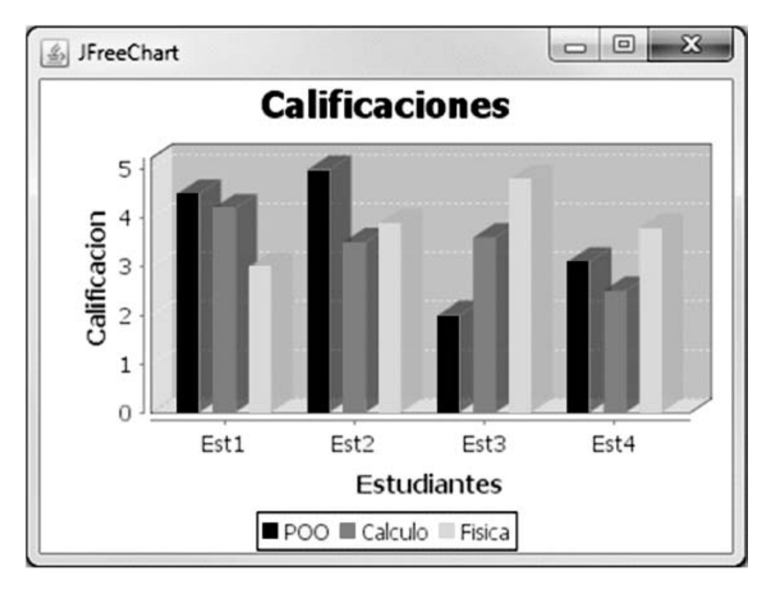


Figura 76. *Bar Chart* 3D

12.4.3 Histogramas

JFreeChart permite la creación de histogramas. En el momento de crear uno de estos *chart* es necesario asignar los datos mediante la clase *HistogramDataSet*. Usando el método *addSeries*, el histograma recibe el nombre de la variable a medir los valores de la variable y la frecuencia.

Ejemplo de Histograma

El siguiente código presenta un ejemplo de creación de un histograma con datos aleatorios. La implementación es la siguiente:

```
package charts.histogram;

import java.awt.BorderLayout;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.WindowConstants;
```

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.statistics.HistogramDataset;
import org.jfree.data.statistics.HistogramType;

public class FHistogram2D extends JFrame {

   public static void main(String[] args) {
       FHistogram2D frame = new FHistogram2D();
       frame.setVisible(true);
   }

   public FHistogram2D() {
       initGUI();
       HistogramDataset dataSet = new HistogramDataset();
       dataSet = createDataSet();
       JFreeChart chart = ChartFactory.createHistogram(
       "Histograma", "Valores", "Frecuencia", dataSet,
       PlotOrientation.VERTICAL, true, true, false);
       ChartPanel panel = new ChartPanel(chart);
       getContentPane().add(panel, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setLayout(new BorderLayout());
       setTitle("JFreeChart");
       setSize(400, 300);
   }

   private HistogramDataset createDataSet() {
       HistogramDataset dataSet = new HistogramDataset();
       Random random = new Random();
       double []valores = new double[100];
       for(int i=0; i<100; i++){
          valores[i]=random.nextDouble()*100;
       }
       dataSet.setType(HistogramType.RELATIVE_FREQUENCY);
       dataSet.addSeries("Variable", valores, 20);
       return dataSet;
   }
}
```


El resultado es el siguiente:

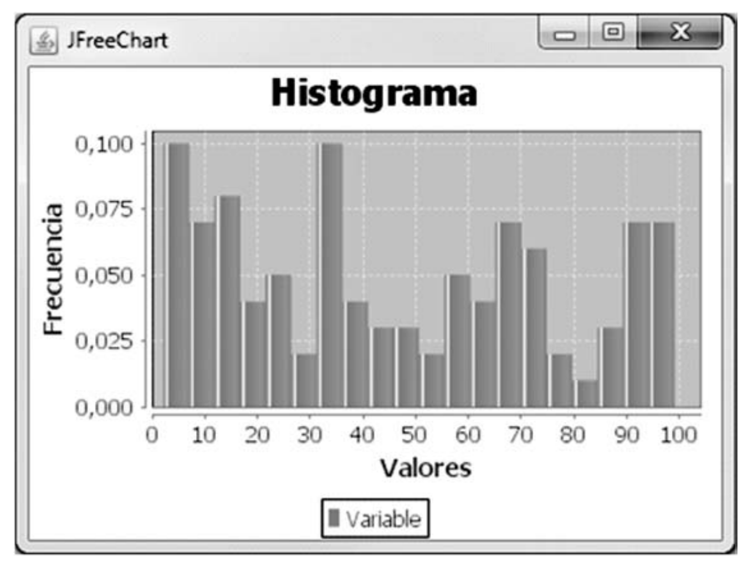


Figura 77. Histograma

12.4.4 Diagramas polares

JFreeChart permite la creación de diagramas polares. En el momento de crear uno de estos *chart* es necesario asignar los datos mediante la clase *XYSeriesCollection,* la cual permite agregar objetos de la clase *XYSeries*. La clase *XTSeries* recibe valores con base en el ángulo y el dato de la serie.

Ejemplo de *Polar Chart*

El siguiente código presenta un ejemplo de creación de un *Polar Chart* con datos aleatorios. La implementación es la siguiente:

```
package charts.polar;

import java.awt.BorderLayout;
import java.util.Random;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

import org.jfree.chart.ChartFactory;
```

```
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

public class FPolar2D extends JFrame {

   public static void main(String[] args) {
       FPolar2D frame = new FPolar2D();
       frame.setVisible(true);
   }

   public FPolar2D() {
       initGUI();
       XYDataset seriesCollection = new XYSeriesCollection();
       seriesCollection = createSeriesCollection();
       JFreeChart chart = ChartFactory.createPolarChart(
       "Polar", seriesCollection, true, true, false);
       ChartPanel panel = new ChartPanel(chart);
       getContentPane().add(panel, BorderLayout.CENTER);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setLayout(new BorderLayout());
       setTitle("JFreeChart");
       setSize(400, 300);
   }

   private XYSeriesCollection createSeriesCollection() {
       XYSeriesCollection seriesCollection =
       new XYSeriesCollection();
       XYSeries series;
       Random random = new Random();
       for(int i=0; i<3; i++){
          series = new XYSeries("Serie "+ (i+1));
          for(int j=0; j<4; j++){
              int Angulo = j*90;
              int valor = (Math.abs(random.nextInt())%10)+1;
              series.add(Angulo, valor);
          }
          seriesCollection.addSeries(series);
       }
       return seriesCollection;
   }
}
```

El resultado es el siguiente:

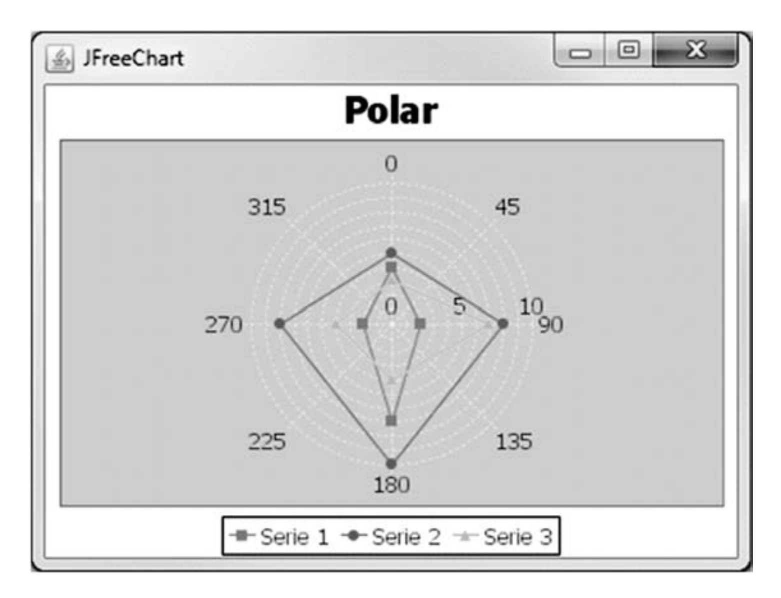


Figura 78. *Polar Chart*

12.5 Ejercicios propuestos

1. Implemente una aplicación que permita dibujar un círculo que posea movimiento con base en el teclado. El círculo debe iniciar en el centro de un área de dibujo (*JPanel*). Si el usuario presiona la tecla "*flecha derecha*" el círculo se desplaza 10 píxeles a la derecha. Si el usuario presiona la tecla "*flecha izquierda*" el círculo se desplaza 10 píxeles a la izquierda. Si el usuario presiona la tecla "*flecha arriba*" el círculo se desplaza 10 píxeles arriba. Si el usuario presiona la tecla "*flecha abajo*" el círculo se desplaza 10 píxeles abajo.

2. Implemente una aplicación que permita dibujar un círculo que persiga el puntero del *mouse*. Esto indica que si el usuario mueve el puntero del *mouse* en el área de dibujo (*JPanel*), el círculo se repinta teniendo como centro las coordenadas del *mouse*.

3. Realice un histograma que contenga los valores del *TRM* del dólar del último año.

CAPÍTULO 13

Acceso a bases de datos

Todo sistema de información posee bases de datos para la persistencia de los datos. Para realizar una conexión es necesario utilizar un componente conocido como *JDBC* (*Java Data Base Connector*) el cual, debe proveer el creador del motor de base de datos que se desee utilizar. El *JDBC* está diseñado para ser independiente de la plataforma e incluso de la base de datos, de esta manera, la aplicación se comunica con el *JDBC* y este con la base de datos. Para manipular la base de datos desde la aplicación en Java es necesario utilizar el lenguaje *SQL* (*Structured Query Language*).

Considerando un motor de base de datos *MySql* se debe descargar el *JDBC* para *Mysql*[1]. El *JDBC* es un componente embebido en un archivo "*jar*". Este archivo debe incluirse en el *build path* del proyecto para poder ser utilizado por la aplicación.

13.1 Conexión a base de datos

Para realizar la conexión es necesario hacer uso de las clases *Conection*, *Statement* y *DriverManager* que hacen parte del paquete **java.sql**. Adicionalmente, se deben conocer los siguientes parámetros:

- *JDBC*. Es necesario conocer la clase que permite que el *JDBC* realice la conexión a la base de datos de acuerdo al motor seleccionado. Para *MySql*, la clase es *com.mysql.jdbc.Driver*.
- Cadena de conexión. Esta cadena contiene información del *JDBC*, servidor, puerto y nombre de base de datos.

[1] http://dev.mysql.com

- Usuario. El usuario debe ser incluido en la conexión para hacer uso de los diferentes servicios del motor de la base de datos.
- Contraseña. Corresponde a la contraseña del usuario anterior.

La siguiente clase muestra una forma de realizar la conexión con una base de datos en motor *MySql*:

Clase *Conexion*

```
package persistencia;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Conexion {
   private Statement statement;
   private Connection connection;
   private String jdbc;
   private String ruta;
   private String usuario;
   private String contra;

   public Conexion(){
       this.connection = null;
       this.statement = null;
       this.jdbc = "com.mysql.jdbc.Driver";
       this.ruta = "jdbc:mysql://localhost:3306/bd";
       this.usuario ="root";
       this.contra ="";
   }

   private void abrirConexion(){
       try {
          Class.forName(this.jdbc);
          this.connection = DriverManager.getConnection(
          this.ruta,this.usuario, this.contra);
          this.statement = this.connection.createStatement();
       }catch (SQLException e){
          e.printStackTrace();
       } catch (ClassNotFoundException e) {
          e.printStackTrace();
       }
   }

   public String ejecutar(String sentencia){
       try {
          this.abrirConexion();
          this.statement.executeUpdate(sentencia);
```

```
            return "Op Exitosa";
        }catch (SQLException e){
            return e.toString();
        }
    }

    public ResultSet consultar(String sentencia){
        ResultSet resultado=null;
        try {
            this.abrirConexion();
            resultado=statement.executeQuery(sentencia);
        }catch (SQLException e){
            e.printStackTrace();
        }
        return resultado;
    }
}
```

En el método *abrirConexión* permite que el atributo *connection* cree una conexión con la base de datos y que el atributo *statement* cree un objeto que permita enviar sentencias *SQL* a la base de datos. Para ello requiere de las siguientes características:

- El método estático *forName* de la clase *Class* permite cargar a la aplicación, el conector de la base de datos. Recibe el parámetro *JDBC* expuesto anteriormente.
- El método estático *getConnection* de la clase *DriverManager* permite establecer una conexión a la base de datos a través del *JDBC*. Recibe como parámetro la cadena de conexión, el usuario y la contraseña.
- El método *createStatement* de la clase *Connection* crea un *statement* a través de la conexión establecida con la base de datos.
- Es necesario realizar el manejo de las siguientes excepciones.
    - *SQLException*. Se produce si no es posible realizar una conexión a la base de datos con el servidor, usuario y contraseña especificados.
    - *ClassNotFoundException.* Se produce si no encuentra la clase *com.mysql.jdbc.Driver* necesaria para establecer la conexión.

El método *ejecutar* permite enviar una sentencia *SQL* que recibe por parámetro, con base en la conexión abierta por medio del método *excecuteUpdate*. Esta sentencia debe ser necesariamente *insert*, *update* o *delete*. Este método debe manejar la excepción *SQLException*, la cual permite controlar casos de excepción como:

- Agregación de un registro cuya llave primaria ya existe.
- Eliminación de un registro que no existe.
- Actualización de un registro que no existe.
- Error de la sintaxis de la sentencia.
- Sentencia con una tabla de la base de datos que no existe.
- Sentencia con una columna que no existe.

El método *consultar* permite enviar una sentencia *SQL* que recibe por parámetro, con base en la conexión abierta por medio del método *excecuteQuery*. Esta sentencia debe ser necesariamente *select*, debido a que retorna información a través de un objeto *Resulset*. Este método debe manejar la excepción *SQLException*, la cual permite controlar casos de excepción como:

- Error de la sintaxis de la sentencia.
- Sentencia con una tabla de la base de datos que no existe.
- Sentencia con una columna que no existe.

13.2 *DAO (Data Access Object)*

El *DAO* es un patrón de diseño de capa de persistencia, que indica que se debe crear una clase por cada entidad existente en el modelo entidad relación de la base de datos. Esta clase debe mapear los atributos de la entidad y proporcionar única y exclusivamente, métodos que construyan las sentencias *SQL* requeridas para la manipulación de los datos. Con base en ello, cada clase *DAO*, implementa métodos altamente cohesionados con cada entidad.

Considerando un almacén que contiene en su base de datos denominada "*BD*", una entidad producto con los atributos *id*, nombre, cantidad y precio. Para la creación de esta base de datos, de la tabla producto y agregación de tres registros se usa las siguientes sentencias en *Mysql*.

Consola *MySql*

```
mysql>create database bd;
mysql>use bd;
mysql>create table producto (
  id int(10) NOT NULL AUTO_INCREMENT,
  nombre varchar(20) NOT NULL,
  cantidad int(10) NOT NULL,
  precio int(10) NOT NULL,
  PRIMARY KEY (`id`)
);

mysql>insert into producto (id, nombre,
cantidad, precio) values
(10, 'Televisor', 10, 1000000),
(20, 'Computador', 50, 1200000),
(30, 'Horno', 10, 500000);
```

La clase *DAO* que mapea dicha entidad es la siguiente:

Clase *ProductoDAO*

```
package persistencia;

public class ProductoDAO {
   private int id;
   private  String nombre;
   private  int cantidad;
   private  long precio;

   public String insertar(){
       return "insert into producto (id,nombre,cantidad,precio)
       values ('"+this.id+"','"+this.nombre+"','"+
       this.cantidad+"','"+this.precio+"')";
   }

   public String consultar(){
       return "select * from producto where id='"+this.id+"'";
   }

   public String actualizar(){
```

```
        return "update producto set nombre='"+this.nombre
        +"', cantidad='"+this.cantidad+"', precio='"+this.precio
        +"' where id='"+this.id+"'";
    }

    public String buscar(String filtro){
        return "select * from producto where nombre like '"+
        filtro+"%'";
    }

    public String eliminar(){
        return "delete from producto where id='"+this.id+"'";
    }
}
```

Esta clase posee cuatro atributos que son *id*, nombre, cantidad y precio; que corresponden a la entidad de la base de datos. Además, posee los siguientes métodos.

- Método *insertar*, que retorna un *String* con la sentencia *SQL* requerida para insertar un producto en la base de datos.
- Método *consultar*, que retorna un *String* con la sentencia *SQL* requerida para consultar un producto de la base de datos con base en su llave primaria que es *id*.
- Método *actualizar*, que retorna un *String* con la sentencia *SQL* requerida para actualizar un producto de la base de datos con base en su llave primaria que es *id*.
- Método *buscar*, que retorna un *String* con la sentencia *SQL* requerida para consultar a través de la sentencia *like* un conjunto de productos de la base de datos. El objetivo de una sentencia de esta característica es obtener un conjunto de productos filtrados por su nombre.
- Método *eliminar*, que retorna un *String* con la sentencia *SQL* requerida para eliminar un producto de la base de datos con base en su llave primaria que es *id*.

Para hacer uso de los *DAO* es necesario crear una capa de lógica, que permita crear instancias de las clases de la capa de persistencia. A través de estas instancias se hace uso de los servicios que contiene la clase conexión y de los servicios de las clases *DAO*.

Con base en el caso del almacén, en donde hay productos, se puede crear en la capa de lógica una clase para este concepto.

La implementación es la siguiente:

Clase *Producto*

```
package logica;

import java.sql.ResultSet;
import java.sql.SQLException;

import persistencia.Conexion;
import persistencia.ProductoDAO;

public class Producto {
   private int id;
   private  String nombre;
   private  int cantidad;
   private  long precio;

   public String insertar() {
       Conexion conexion = new Conexion();
       ProductoDAO productoDAO = new ProductoDAO(this.id,
       this.nombre, this.cantidad, this.precio);
       return conexion.ejecutar(productoDAO.insertar());

   }

   public String eliminar() {
       Conexion conexion = new Conexion();
       ProductoDAO productoDAO = new ProductoDAO(this.id);
       return conexion.ejecutar(productoDAO.eliminar());

   }

   public boolean consultar() {
       Conexion conexion = new Conexion();
       ProductoDAO productoDAO = new ProductoDAO(this.id);
       ResultSet resultado = conexion.consultar(
       productoDAO.consultar());
       try {
          if (resultado.next()){
          this.nombre=resultado.getString("nombre");
          this.cantidad=resultado.getInt("cantidad");
          this.precio=resultado.getLong("precio");
```

```
            return true;
        }else{
            return false;
        }
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}

    public String actualizar() {
        Conexion conexion = new Conexion();
        ProductoDAO productoDAO = new ProductoDAO(this.id,
        this.nombre, this.cantidad, this.precio);
        return conexion.ejecutar(productoDAO.actualizar());

    }

    public String[][] buscar(String filtro){
        Conexion con = new Conexion();
        ProductoDAO productoDAO = new ProductoDAO();
        ResultSet resultado=con.consultar(
        productoDAO.buscar(filtro));
        String [][] datos=null;
        try {
            resultado.last();
            datos=new String[resultado.getRow()][4];
            resultado.beforeFirst();
            int i=0;
            while(resultado.next()){
                datos[i][0]=resultado.getString("id");
                datos[i][1]=resultado.getString("nombre");
                datos[i][2]=resultado.getString("cantidad");
                datos[i][3]=resultado.getString("precio");
                i++;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return datos;
    }
}
```

La clase *Producto* contiene tres constructores sobrecargados con el fin de suministrar la información necesaria según el acceso a la base de datos que se desee realizar.

El método insertar crea una instancia llamada *Conexión* de la clase *Conexión* y otra instancia llamada producto *DAO*, de la clase *ProductoDAO* enviando por el constructor toda la información del producto a través de sus atributos. Con el objeto *conexión* se hace el llamado al método *ejecutar* en donde se envía por parámetro la sentencia *SQL*, que retorna el método *insertar* de la clase *ProductoDAO*. Este método retorna un *String* que contiene el mensaje "*Op Exitosa*", en caso de realizar correctamente el proceso o un mensaje generado por el manejo de excepción en caso que falle el proceso.

El método *eliminar* funciona de la misma forma que el método *insertar*, excepto por el hecho que solo envía la identificación por parámetro en el momento de crear la instancia de la clase *ProductoDAO* y utiliza el método eliminar de esta clase con el fin de ejecutar la sentencia *SQL* requerida para dicho proceso.

El método *consultar*, crea una instancia llamada *Conexión* de la clase *Conexión* y otra instancia llamada *productoDAO*, de la clase *ProductoDAO*, enviando por el constructor solamente la identificación del producto. Con el objeto *Conexión* hace el llamado al método consultar en donde se envía por parámetro la sentencia *SQL*, que retorna el método consultar de la clase *ProductoDAO*. El método *consultar* de la clase *Conexión*, retorna un *Resulset,* el cual contiene la información del producto consultado. El método *next* de la clase *Resulset* permite avanzar en el conjunto de registros, retornado al realizar la consulta. Además, este método retorna verdadero, si hay registros al avanzar y retorna falso, si no hay registros al avanzar. Debido a que la consulta se hace a través de la llave primaria se asegura que hay un registro, en caso de que exista el producto o cero registros en caso que no exista. Por tal motivo, la sentencia *if* (*resultado.next()*), permite identificar si hay o no registro y en caso de haberlo, permite asignar a los atributos de la clase la información obtenida de la base de datos con base en el método *getString*, *getInt* y *getLong*; los cuales reciben como parámetro el nombre de la columna en la tabla consultada de base de datos.

El método *actualizar* funciona de la misma forma que el método *insertar*, excepto por el hecho que utiliza el método *actualizar* de la

clase *ProductoDAO,* con el fin de ejecutar la sentencia *SQL* requerida para dicho proceso.

El método *buscar*, crea una instancia llamada *Conexión* de la clase *Conexión* y otra instancia llamada *productoDAO* de la clase *ProductoDAO*. Con el objeto *Conexión* hace el llamado al método *consultar*, en donde se envía por parámetro la sentencia *SQL*, que retorna el método *buscar* de la clase *ProductoDAO* que recibe un *String* que contiene un filtro. El método *consultar* de la clase *Conexión* retorna un *Resulset*, el cual contiene la información del conjunto de productos consultados. A diferencia del método *consultar*, el *Resulset* puede tener múltiples registros. Por tal motivo, la sentencia *while* (*resultado.next()*) permite a través de un proceso iterativo identificar si hay o no más registros, y en caso de haberlos permite asignar a una matriz la información obtenida de la base de datos con base en el método *getString*, *getInt* y *getLong* los cuales reciben como parámetro el nombre de la columna en la tabla consultada de base de datos. Los métodos *last* y *beforeFirst* se utilizan para conocer la cantidad de registros obtenidos antes de iniciar el proceso iterativo, con el fin de asignar el tamaño apropiado a la matriz. Este método hace el retorno de dicha matriz.

La capa de *Presentación* en el caso del *almacén* debe incluir los *frames* y páneles necesarios para manipular las clases de la capa de *Lógica*. Para demostrar el funcionamiento se presentan los siguientes *frames*:

- *FPrincipal*. Este *frame* contiene un menú que provee servicios de gestión de productos como agregar, consultar y buscar.
- Agregar Producto. Este *frame* contiene el formulario para agregar un producto a la base de datos.
- Consultar Producto. Este *frame* contiene un formulario en el cual se visualiza la información consultada de la base de datos. Además, provee mecanismos para la actualización de los datos.
- Buscar Producto. Este *frame* contiene una tabla en donde se presentan los registros encontrados basados en un filtro proporcionado.

La implementación de la capa de presentación es la siguiente:

Clase *FPrincipal*

```
package presentacion;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.WindowConstants;

public class FPrincipal extends JFrame {
   private JMenuBar menuBar;
   private JMenu menuProducto;
   private JDesktopPane desktopPane;
   private JMenuItem menuItemBuscarProducto;
   private JMenuItem menuItemConsultarProducto;
   private JMenuItem menuItemAgregarProducto;

   public FPrincipal() {
       initGUI();
   }

   public static void main(String[] args) {
       FPrincipal frame = new FPrincipal();
       frame.setVisible(true);
   }

   private void initGUI() {
       setDefaultCloseOperation(
       WindowConstants.DISPOSE_ON_CLOSE);
       setTitle("Gestion de Productos");
       {
          desktopPane = new JDesktopPane();
          getContentPane().add(desktopPane);
       }
       {
          menuBar = new JMenuBar();
          setJMenuBar(menuBar);
          {
              menuProducto = new JMenu();
              menuBar.add(menuProducto);
              menuProducto.setText("Producto");
              {
                 menuItemAgregarProducto = new JMenuItem();
                 menuProducto.add(menuItemAgregarProducto);
                 menuItemAgregarProducto.setText("Agregar");
```

```
                menuItemAgregarProducto.addActionListener(
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        menuItemAgregarProductoActionPerformed(evt);
                    }
                });
            }
            {
                menuItemConsultarProducto = new JMenuItem();
                menuProducto.add(menuItemConsultarProducto);
                menuItemConsultarProducto.setText("Consultar");
                menuItemConsultarProducto.addActionListener(
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        menuItemConsultarProductoActionPerformed(evt);
                    }
                });
            }
            {
                menuItemBuscarProducto = new JMenuItem();
                menuProducto.add(menuItemBuscarProducto);
                menuItemBuscarProducto.setText("Buscar");
                menuItemBuscarProducto.addActionListener(
                new ActionListener() {
                    public void actionPerformed(ActionEvent evt) {
                        menuItemBuscarProductoActionPerformed(evt);
                    }
                });
            }
        }
    }
    setSize(400, 300);
}

protected void menuItemAgregarProductoActionPerformed(
ActionEvent evt) {
    FIAgregarProducto frame = new FIAgregarProducto();
    this.desktopPane.add(frame);

}

private void menuItemConsultarProductoActionPerformed(
ActionEvent evt) {
    FIConsultarProducto frame = new FIConsultarProducto();
    this.desktopPane.add(frame);
}

private void menuItemBuscarProductoActionPerformed(
ActionEvent evt) {
    FIBuscarProducto frame = new FIBuscarProducto();
    this.desktopPane.add(frame);
}
}
```

Este *frame* provee un menú para realizar las diferentes operaciones de uso de la base de datos. El resultado es el siguiente:

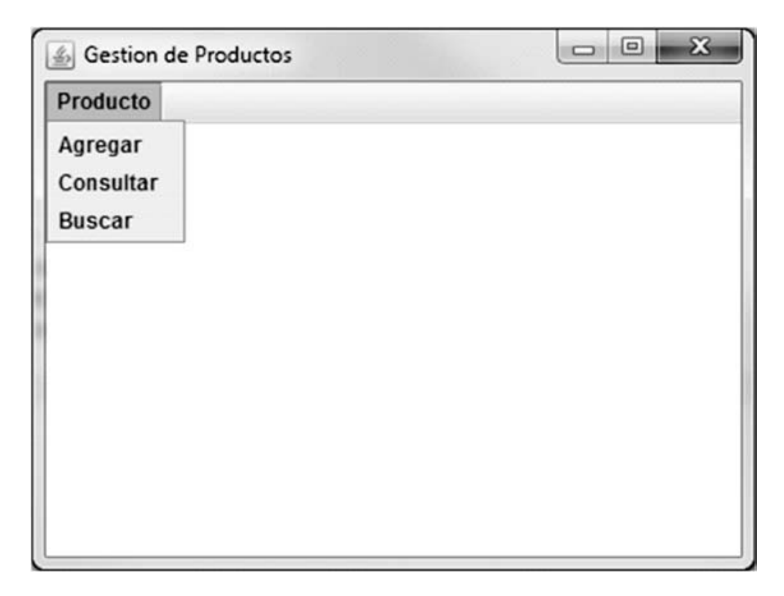


Figura 79. *Frame* de gestión de productos

Clase *FIAgregarProducto*

```
package presentacion;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JInternalFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import logica.Producto;

public class FIAgregarProducto extends JInternalFrame {
   private JLabel labelNombre;
   private JTextField textNombre;
   private JLabel labelCantidad;
   private JButton buttonLimpiar;
   private JButton buttonAgregar;
   private JTextField textPrecio;
```

```
private JLabel labelPrecio;
private JTextField textCantidad;
private JTextField textId;
private JLabel labelId;

public FIAgregarProducto() {
    initGUI();
}

private void initGUI() {
    setTitle("Agregar Producto");
    setVisible(true);
    setClosable(true);
    getContentPane().setLayout(new GridLayout(5, 2, 0, 0));
    {
       labelId = new JLabel();
       getContentPane().add(labelId);
       labelId.setText("Id");
    }
    {
       textId = new JTextField();
       getContentPane().add(textId);
    }
    {
       labelNombre = new JLabel();
       getContentPane().add(labelNombre);
       labelNombre.setText("Nombre");
    }
    {
       textNombre = new JTextField();
       getContentPane().add(textNombre);
    }
    {
       labelCantidad = new JLabel();
       getContentPane().add(labelCantidad);
       labelCantidad.setText("Cantidad");
    }
    {
       textCantidad = new JTextField();
       getContentPane().add(textCantidad);
    }
    {
       labelPrecio = new JLabel();
       getContentPane().add(labelPrecio);
       labelPrecio.setText("Precio");
    }
    {
       textPrecio = new JTextField();
       getContentPane().add(textPrecio);
    }
```

```
        {
           buttonAgregar = new JButton();
           getContentPane().add(buttonAgregar);
           buttonAgregar.setText("Agregar");
           buttonAgregar.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent evt) {
                  buttonAgregarActionPerformed(evt);
               }
           });
        }
        {
           buttonLimpiar = new JButton();
           getContentPane().add(buttonLimpiar);
           buttonLimpiar.setText("Limpiar");
           buttonLimpiar.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent evt) {
                  buttonLimpiarActionPerformed(evt);
               }
           });
        }
        setSize(250,150);
    }

    private void buttonAgregarActionPerformed(ActionEvent evt) {
        Producto producto = new Producto(Integer.parseInt(
        this.textId.getText()), this.textNombre.getText(),
        Integer.parseInt(this.textCantidad.getText(
        )),Long.parseLong(this.textPrecio.getText()));
        JOptionPane.showMessageDialog(this, producto.agregar(
        ), "Agregar", JOptionPane.INFORMATION_MESSAGE);
        buttonLimpiarActionPerformed(evt);
    }

    private void buttonLimpiarActionPerformed(ActionEvent evt) {
        textId.setText("");
        textNombre.setText("");
        textCantidad.setText("");
        textPrecio.setText("");
    }
}
```

Este *frame* contiene un formulario que permite ingresar la información del producto a la base de datos a través del botón Agregar. Si ingresa el producto correctamente presenta un cuadro de mensaje con el texto "*Op Exitosa*".

El *frame* se presenta de la siguiente forma:

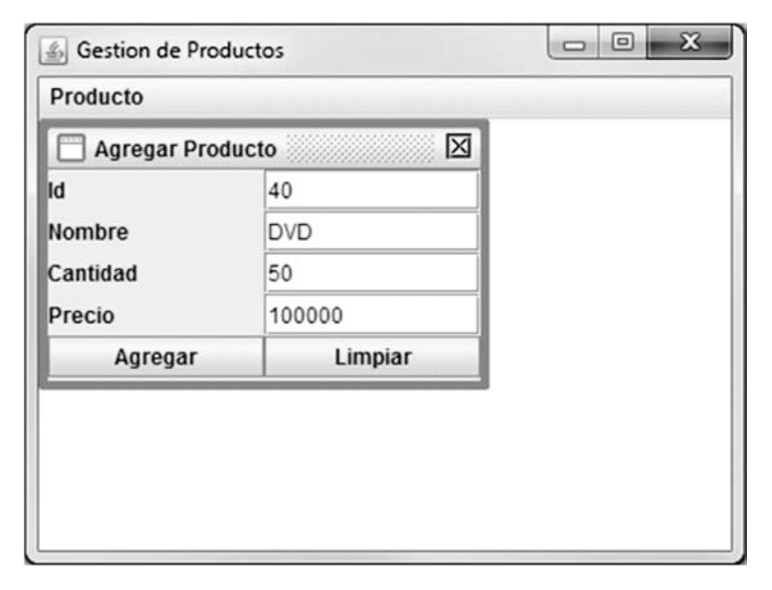


Figura 80. *Frame* de agregar productos

Los resultados en *Mysql* se pueden verificar con la siguiente sentencia:

Consola *MySql*

```
mysql> select * from producto;
+----+------------+----------+---------+
| id | nombre     | cantidad | precio  |
+----+------------+----------+---------+
| 10 | Televisor  |       10 | 1000000 |
| 20 | Computador |       50 | 1200000 |
| 30 | Horno      |       10 |  500000 |
| 40 | DVD        |       50 |  100000 |
+----+------------+----------+---------+
4 rows in set (0.00 sec)
```

Clase *FIConsultarProducto*

```
package presentacion;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;
import javax.swing.JInternalFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import logica.Producto;

public class FIConsultarProducto extends JInternalFrame {
   private JLabel labelNombre;
   private JTextField textNombre;
   private JLabel labelCantidad;
   private JButton buttonActualizar;
   private JButton buttonConsultar;
   private JTextField textPrecio;
   private JLabel labelPrecio;
   private JTextField textCantidad;
   private JTextField textId;
   private JLabel labelId;

   public FIConsultarProducto() {
       initGUI();
   }

   private void initGUI() {
       setTitle("Consultar Producto");
       setVisible(true);
       setClosable(true);
       getContentPane().setLayout(new GridLayout(5, 2, 0, 0));
       {
          labelId = new JLabel();
          getContentPane().add(labelId);
          labelId.setText("Id");
       }
       {
          textId = new JTextField();
          getContentPane().add(textId);
       }
       {
          labelNombre = new JLabel();
          getContentPane().add(labelNombre);
          labelNombre.setText("Nombre");
       }
       {
          textNombre = new JTextField();
          getContentPane().add(textNombre);
       }
       {
          labelCantidad = new JLabel();
          getContentPane().add(labelCantidad);
```

```
        labelCantidad.setText("Cantidad");
    }
    {
        textCantidad = new JTextField();
        getContentPane().add(textCantidad);
    }
    {
        labelPrecio = new JLabel();
        getContentPane().add(labelPrecio);
        labelPrecio.setText("Precio");
    }
    {
        textPrecio = new JTextField();
        getContentPane().add(textPrecio);
    }
    {
        buttonConsultar = new JButton();
        getContentPane().add(buttonConsultar);
        buttonConsultar.setText("Consultar");
        buttonConsultar.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                buttonConsultarActionPerformed(evt);
            }
        });
    }
    {
        buttonActualizar = new JButton();
        getContentPane().add(buttonActualizar);
        buttonActualizar.setText("Actualizar");
        buttonActualizar.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                buttonActualizarActionPerformed(evt);
            }
        });
    }
    setSize(250,150);
}

private void buttonConsultarActionPerformed(ActionEvent evt)
{
    Producto producto = new
    Producto(Integer.parseInt(this.textId.getText()));
    if(producto.consultar()){
        this.textNombre.setText(producto.getNombre());
        this.textCantidad.setText(String.valueOf(
        producto.getCantidad()));
        this.textPrecio.setText(String.valueOf(
        producto.getPrecio()));
    }else{
        JOptionPane.showMessageDialog(this,
```

```
            "No hay resultados", "Consultar",
            JOptionPane.INFORMATION_MESSAGE);
        }
    }

    private void buttonActualizarActionPerformed(ActionEvent evt) {
        Producto producto = new Producto
        (Integer.parseInt(this.textId.getText()),
        this.textNombre.getText(), Integer.parseInt
        (this.textCantidad.getText()),Long.parseLong
        (this.textPrecio.getText()));
        JOptionPane.showMessageDialog(this, producto.actualizar(),
        "Actualizar", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Consultando el producto con código 10, el resultado es el siguiente:

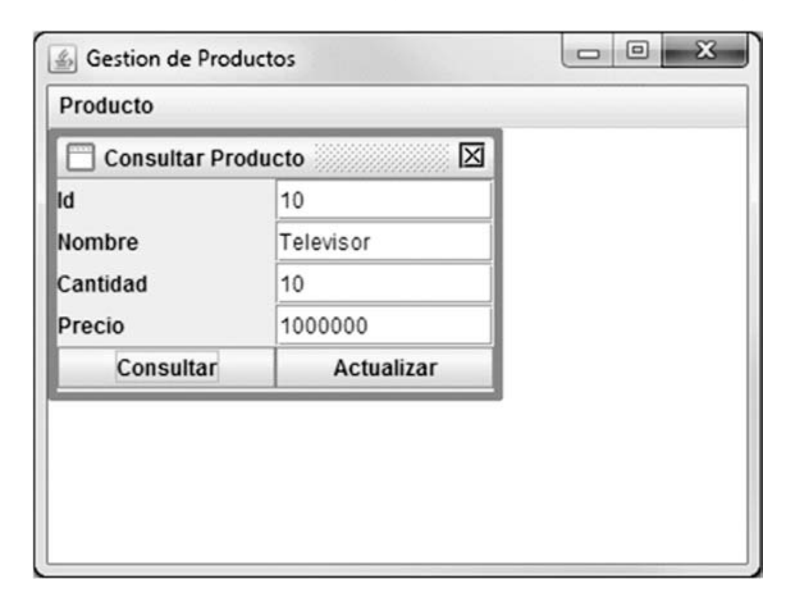


Figura 81. *Frame* de consultar productos

Clase *FIBuscarProducto*

```
package presentacion;

import java.awt.BorderLayout;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
```

```
import javax.swing.JInternalFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;

import logica.Producto;

public class FIBuscarProducto extends JInternalFrame {
   private JTextField textFiltro;
   private JTable tableResultados;
   private JScrollPane scrollPaneResultados;

   public FIBuscarProducto() {
       initGUI();
   }

   private void initGUI() {
       setTitle("Buscar Producto");
       setVisible(true);
       setClosable(true);
       getContentPane().setLayout(new BorderLayout());
       {
          textFiltro = new JTextField();
          getContentPane().add(textFiltro, BorderLayout.NORTH);
          textFiltro.addKeyListener(new KeyAdapter() {
              public void keyReleased(KeyEvent evt) {
                 textFiltroKeyReleased(evt);
              }
          });
       }
       {
          scrollPaneResultados = new JScrollPane();
          getContentPane().add(scrollPaneResultados,
          BorderLayout.CENTER);
          {
              tableResultados = new JTable();
              scrollPaneResultados.setViewportView(
              tableResultados);
              tableResultados.setPreferredSize(
              new java.awt.Dimension(250, 100));
          }
       }
       setSize(250,150);
   }

   private void textFiltroKeyReleased(KeyEvent evt) {
       Producto producto = new Producto();
       String [][] datos = producto.buscar(
```

```
        this.textFiltro.getText());
        String [] titulos = new String[] { "Id", "Nombre",
        "Cantidad", "Precio"};
        TableModel tableModelResultados = new
        DefaultTableModel(datos, titulos);
        tableResultados.setModel(tableModelResultados);
        tableResultados.setPreferredSize(
        new java.awt.Dimension(350,datos.length*16));
        TableRowSorter ordenador=new TableRowSorter(
        tableModelResultados);
        this.tableResultados.setRowSorter(ordenador);
    }
}
```

Este *frame* contiene una tabla que permite visualizar todos los productos. Adicionalmente, tiene un cuadro de texto que permite filtrar la información de productos de acuerdo con el texto incluido, de tal forma que visualizará en la tabla todos los resultados en donde el nombre del producto inicie por el texto colocado.

El resultado es el siguiente:

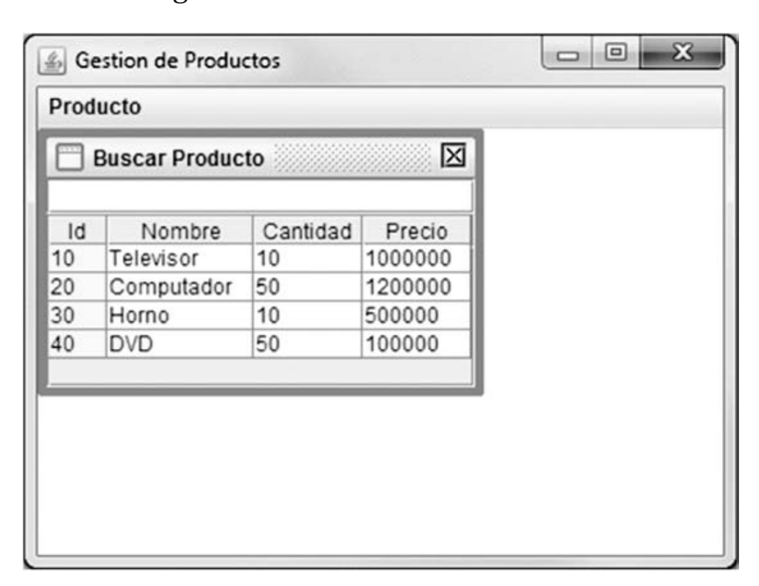


Figura 82. *Frame* de buscar productos

Si se coloca un filtro como la letra "*t*", el resultado es el siguiente:

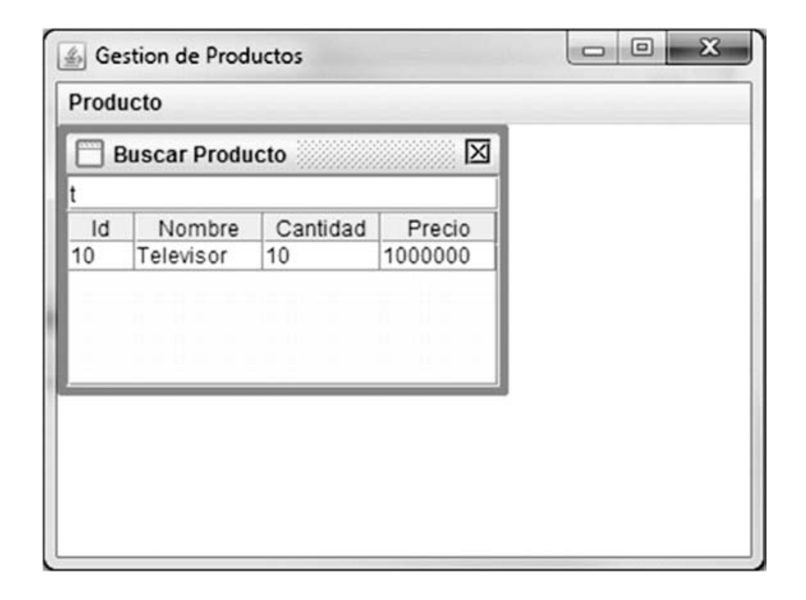


Figura 83. *Frame* de buscar productos con filtro

Además, haciendo clic en el nombre de la columna, se ordenan los resultados gracias al uso de la clase *TableRowSorter*.

13.3 Ejercicios propuestos

1. Implemente una aplicación, que permita simular un sistema de registro académico que contenga las siguientes características:

   a. Gestión de Administradores. Un administrador podrá ingresar nuevos administradores en el sistema.

   b. Gestión de Estudiantes. Un administrador podrá ingresar nuevos estudiantes en el sistema. También podrá consultarlos y actualizarlos.

   c. Gestión de Profesores. Un administrador podrá

ingresar nuevos profesores en el sistema. También podrá consultarlos y actualizarlos.

d. Gestión de Cursos. Un administrador podrá ingresar nuevos cursos en el sistema. También podrá consultarlos y actualizarlos. Al ingresar un curso, el administrador deberá indicar el profesor que impartirá dicho curso.

e. Gestión de Inscripción. Un estudiante podrá inscribir un curso. Un profesor podrá digitar notas de un curso que le haya sido asignado.

2. Implemente una aplicación que permita simular un sistema de supermercado, que contenga las siguientes características:

   a. Gestión de Administradores. Un administrador podrá ingresar nuevos administradores en el sistema.

   b. Gestión de Clientes. Un administrador podrá ingresar nuevos clientes en el sistema. También podrá consultarlos y actualizarlos.

   c. Gestión de Cajeros. Un administrador podrá ingresar nuevos cajeros en el sistema. También podrá consultarlos y actualizarlos.

   d. Gestión de Productos. Un administrador podrá ingresar nuevos productos en el sistema. También podrá consultarlos y actualizarlos.

   e. Gestión de Ventas. Un vendedor podrá realizar una venta que incluye un conjunto de productos. Para que un producto pueda ser vendido, la cantidad de unidades de un producto existentes en el sistema, debe ser igual o superior a la cantidad de unidades de un producto que intenta venderse.

CAPÍTULO 14

Modelo Vista Controlador

El patrón *Modelo Vista Controlador* (*MVC*) es un patrón de diseño para interfaces gráficas de usuario que divide la aplicación en tres diferentes responsabilidades que son:

- Modelo. Contiene los objetos de dominio o estructuras de dato que representan la lógica de negocio y contienen el estado de la aplicación.

- Vista. Hace referencia a los componentes gráficos que son la salida de información hacia el usuario de la aplicación. La vista se actualiza de acuerdo el estado del modelo, lo que indica que la vista permanentemente observa al modelo.

- Controlador. Son controles que se encuentran disponibles al usuario mediante la interfaz gráfica. Estos controles los usa el usuario a través del evento como hacer clic a un botón, presionar una tecla, mover el *mouse*, etc. El controlador es quien altera el modelo en el momento de que el usuario realiza el evento.

Entonces, el patrón *MVC* es una división de objetos de dominio que modelan la abstracción del mundo real que la aplicación representa, con objetos de presentación que proporcionan una interfaz gráfica, por la cual se le presenta la información al usuario.

La Figura 84 muestra cómo interactúan los tres componentes del patrón.

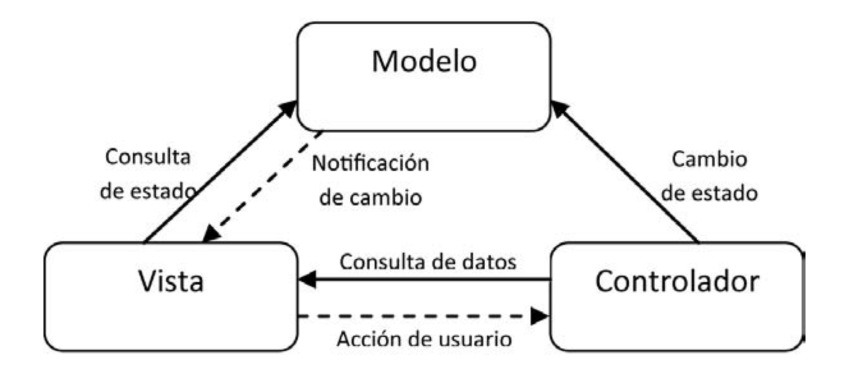


Figura 84. Modelo Vista Controlador

En la figura anterior, la flecha continua determina invocación de métodos, mientras que la flecha punteada determina eventos del usuario. Lo anterior, indica que la vista y el controlador poseen objetos que son instancias de las clases del modelo, además, el controlador (que son eventos que se realizan mediante interfaz gráfica como botones) puede acceder a los elementos de la vista con el fin de capturar información, que el usuario proporciona para llevar a cabo determinadas acciones en el modelo. El modelo notifica a la vista si se han producido cambios, como consecuencia de la invocación de métodos de cambio de estado por parte del controlador. Esta notificación se hace de forma indirecta. Para esta invocación indirecta se usa el patrón observador.

14.1 Patrón observador

Es un patrón de diseño que define una dependencia entre objetos. El objetivo es notificar a los objetos dependientes si el objeto independiente realiza un cambio. Este patrón es de comportamiento, lo que implica que su uso está relacionado con algoritmos de funcionamiento y asignación de responsabilidades. Para implementar el patrón observador en Java es necesario que el modelo herede la clase "*Observable*" y que la vista implemente la interfaz "*Observer*". Esta interfaz requiere que la vista sobre escriba el método *Update,* el cual trae por parámetro un objeto que corresponde al modelo.

14.2 Ejemplo de patrón MVC

Con el fin de ilustrar el patrón *MVC*, usando el patrón observador, se utiliza como caso de estudio el juego *Pacman*. Este juego contiene el siguiente diagrama de clases:

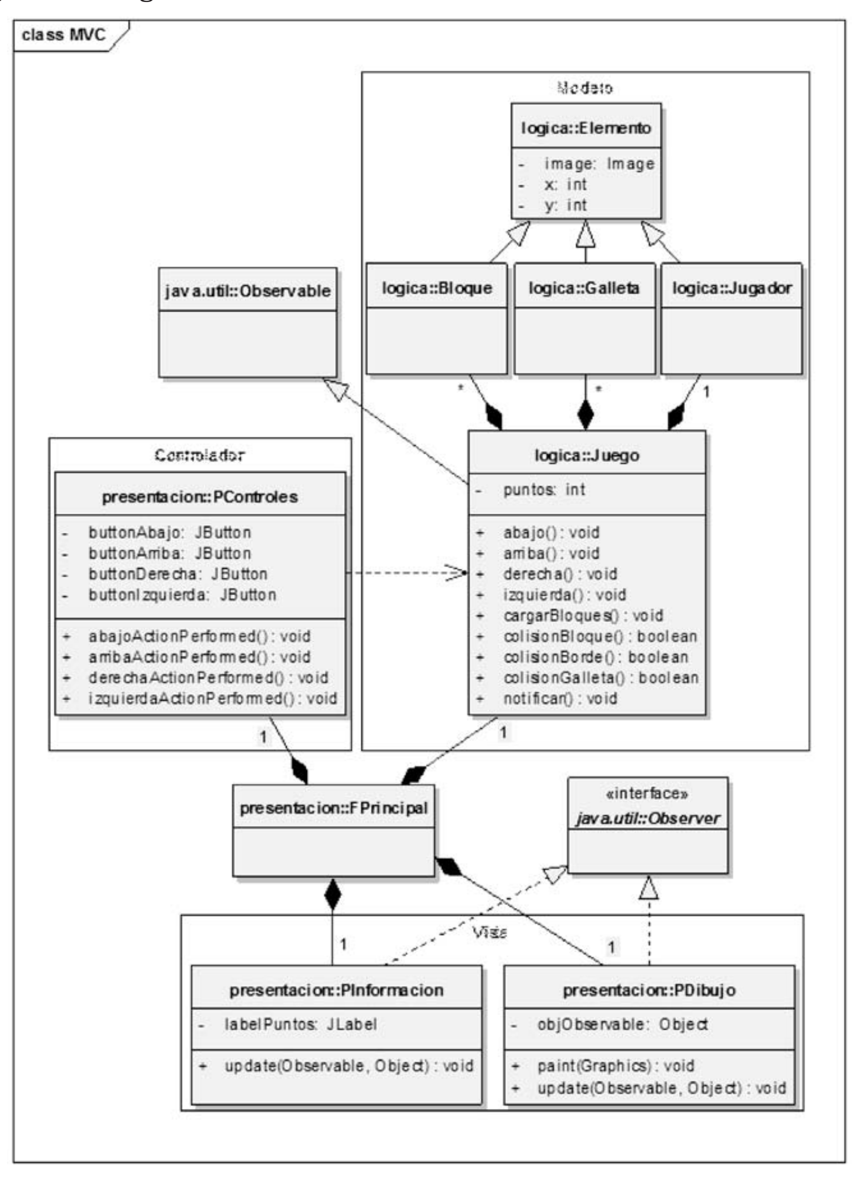


Figura 85. Diagrama de clases del juego Pacman basado en MVC

En el diagrama de clases, las clases que se encuentran en el paquete de lógica hacen referencia al modelo. La clase *Juego* hereda de la clase *Observable* y contiene instancias de las demás clases que son *Bloque*, *Galleta* y *Jugador;* las cuales heredan de *Elemento*. La vista está conformada por las clases *PInformacion*, la cual permite visualizar los puntos del juego y *PDibujo*, la cual permite visualizar gráficamente el estado del juego. Estas dos clases implementan la interfaz *Observer,* la que permite que la vista pueda recibir notificaciones de cambio del modelo. El controlador está conformado por la clase *PControles* que tiene una instancia de la clase *Juego*, con el fin de acceder a los métodos que modifican el estado.

Debido a que la clase *Juego* hereda de *Observable*, se puede asignar la instancia de la clase *Juego* a los objetos que son la vista, es decir, los objetos que implementan la interfaz *Observer*. Este proceso se lleva a cabo mediante el método "*addObserver*". El modelo notifica a la vista mediante el método "*notifyObservers*". Las clases de la vista conocen la notificación del modelo a través de la invocación del método sobre escribible "*update*".

La implementación del juego de Pacman mediante el patrón *MVC* con base en el patrón observador es la siguiente:

Clase *Juego*

```
package logica;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Observable;
import java.util.Properties;

public class Juego extends Observable{
   public final static int DETENIDO=-1;
   public final static int DERECHA=0;
   public final static int ARRIBA=1;
```

```
public final static int IZQUIERDA=2;
public final static int ABAJO=3;

private Jugador comeGalletas;
private ArrayList<Galleta> galletas;
private ArrayList<Bloque> bloques;
private int puntos;

public Juego(){
    this.comeGalletas = new Jugador(0, 0);
    this.galletas = new ArrayList<Galleta>();
    this.bloques = new ArrayList<Bloque>();
    this.cargarBloques();
    for(int i=0; i<10; i++){
       for(int j=0; j<7; j++){
           if(!existeBloque(i, j)){
              this.galletas.add(new Galleta(i, j));
           }
       }
    }
    this.galletas.remove(0);
}

public Jugador getComeGalletas() {
    return comeGalletas;
}

public ArrayList<Galleta> getGalletas() {
    return galletas;
}

public ArrayList<Bloque> getBloques() {
    return bloques;
}

public int getPuntos() {
    return puntos;
}

private void cargarBloques(){
    File archivo = new File("res/bloques.properties");
    try {
       FileInputStream fis = new FileInputStream(archivo);
       Properties archivoPropiedades = new Properties();
       archivoPropiedades.load(fis);
       for(int i=1; i<=24; i++){
           String dato = archivoPropiedades.getProperty(
           "bloque"+i);
```

```
                String datos[] = dato.split(“,”);
                int y = Integer.parseInt(datos[0]);
                int x = Integer.parseInt(datos[1]);
                this.bloques.add(new Bloque(x, y));
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private boolean existeBloque(int x, int y) {
        for(Bloque b : this.bloques){
            if(b.getX()==x && b.getY()==y){
                return true;
            }
        }
        return false;
    }

    public void arriba(){
        if(!this.colisionBorde(Juego.ARRIBA) &&
        !this.colisionBloque(Juego.ARRIBA)){
            this.comeGalletas.arriba();
            this.colisionGalleta();
        }
        this.notificar();
    }

    public void abajo(){
        if(!this.colisionBorde(Juego.ABAJO) &&
        !this.colisionBloque(Juego.ABAJO)){
            this.comeGalletas.abajo();
            this.colisionGalleta();
        }
        this.notificar();
    }

    public void derecha(){
        if(!this.colisionBorde(Juego.DERECHA) &&
        !this.colisionBloque(Juego.DERECHA)){
            this.comeGalletas.derecha();
            this.colisionGalleta();
        }
        this.notificar();
    }
```

```
public void izquierda(){
    if(!this.colisionBorde(Juego.IZQUIERDA) &&
    !this.colisionBloque(Juego.IZQUIERDA)){
       this.comeGalletas.izquierda();
       this.colisionGalleta();
    }
    this.notificar();
}

private boolean colisionBloque(int direccion){
    int x = this.comeGalletas.getX();
    int y = this.comeGalletas.getY();
    for(Bloque bloque : this.bloques){
       if( (direccion==Juego.DERECHA &&
       bloque.getX()==x+1 && bloque.getY()==y) ||
           (direccion==Juego.ABAJO && bloque.getX()==x &&
           bloque.getY()==y+1) ||
           (direccion==Juego.IZQUIERDA && bloque.getX(
           )==x-1 && bloque.getY()==y) ||
           (direccion==Juego.ARRIBA && bloque.getX(
           )==x && bloque.getY()==y-1)) {
           return true;
       }
    }
    return false;
}

private boolean colisionBorde(int direccion){
    int x = this.comeGalletas.getX();
    int y = this.comeGalletas.getY();
    if( (direccion==Juego.DERECHA && x==9) ||
       (direccion==Juego.ABAJO && y==6) ||
       (direccion==Juego.IZQUIERDA && x==0) ||
       (direccion==Juego.ARRIBA && y==0)){
       return true;
    }else{
       return false;
    }
}

public void colisionGalleta(){
    Iterator<Galleta> iterator = this.galletas.iterator();
    while (iterator.hasNext()){
       Galleta galleta = iterator.next();
       if(galleta.getX()==this.comeGalletas.getX() &&
       galleta.getY()==this.comeGalletas.getY()){
           this.puntos++;
           iterator.remove();
```

```
            return;
         }
      }
   }

   public void notificar(){
      this.setChanged();
      this.notifyObservers(this);
   }
}
```

Esta clase tiene los siguientes métodos:

- Constructor. Inicializa el jugador, las galletas y los bloques.
- El método cargar bloques crea bloques con base en un archivo de propiedades que tiene la siguiente información:

Archivo "*bloques.properties*"

`bloque1=1,1`
`bloque2=1,2`
`bloque3=1,3`
`bloque4=1,4`
`bloque5=1,5`
`bloque6=1,6`
`bloque7=1,7`
`bloque8=1,8`
`bloque9=2,1`
`bloque10=2,8`
`bloque11=3,3`
`bloque12=3,4`
`bloque13=3,5`
`bloque14=3,6`
`bloque15=4,1`
`bloque16=4,8`
`bloque17=5,1`
`bloque18=5,2`
`bloque19=5,3`
`bloque20=5,4`
`bloque21=5,5`
`bloque22=5,6`
`bloque23=5,7`
`bloque24=5,8`

- El método *existeBloque* verifica que existe un bloque en la coordenada *x*, y recibida por parámetro.
- Los métodos arriba, abajo, derecha e izquierda, realizan un movimiento del jugador modificando su estado y verifica si consume una galleta.
- Los métodos *colisionBorde*, *colisionBloque*, *colisionGalleta*, verifican si el movimiento hace que toque el borde o un bloque o si toca una galleta. En el caso de que sean los dos primeros, el movimiento no se lleva a cabo. En el caso de que tome una galleta se incrementa el atributo puntos.
- El método *notificar*, notifica a los observadores que el estado ha cambiado mediante el uso del método *notifyObservers.*

Clase *Elemento*

```
package logica;

import java.awt.Image;
import java.awt.Toolkit;

public class Elemento {
   protected int x;
   protected int y;
   protected Image image;

   public Elemento (int x, int y, String imagen){
       this.x = x;
       this.y = y;
       this.image = Toolkit.getDefaultToolkit().getImage(imagen);
   }

   public int getX(){
       return this.x;
   }

   public void setX(int x){
       this.x=x;
   }

   public int getY(){
       return this.y;
   }
```

```
   public void setY(int y){
       this.y=y;
   }

   public Image getImagen(){
       return this.image;
   }
}
```

Clase *Jugador*

```
package logica;

public class Jugador extends Elemento{

   public Jugador (int x, int y){
       super(x, y, "img/pacman.gif");
   }

   public void arriba(){
       this.y-=1;
   }

   public void abajo(){
       this.y+=1;
   }

   public void derecha(){
       this.x+=1;
   }

   public void izquierda(){
       this.x-=1;
   }
}
```

Clase *Bloque*

```
package logica;

public class Bloque extends Elemento{

   public Bloque (int x, int y){
       super(x, y, "img/bloque.png");
       }
}
```

Clase *Galleta*

```
package logica;

public class Galleta extends Elemento{

   public Galleta (int x, int y){
       super(x, y, "img/galleta.gif");
   }
}
```

Clase *FPrincipal*

```
package presentacion;

import logica.*;
import javax.swing.*;
import java.awt.*;

public class FPrincipal extends JFrame{
   private Juego juegoPacman;
   private PDibujo panelDibujo;
   private PControles panelControles;
   private PInformacion panelInformacion;

   public FPrincipal (){
       this.setDefaultCloseOperation(WindowConstants.
       DISPOSE_ON_CLOSE);
       this.setResizable(false);
       this.setSize(205,300);
       this.setTitle("Pacman");
       this.juegoPacman = new Juego();
       this.setLayout(new BorderLayout());
       this.panelDibujo = new PDibujo();
       this.getContentPane().add(this.panelDibujo,
       BorderLayout.CENTER);
       this.panelControles = new PControles(this.juegoPacman) ;
       this.getContentPane().add(this.panelControles,
       BorderLayout.SOUTH);
       this.panelInformacion = new PInformacion() ;
       this.getContentPane().add(this.panelInformacion,
       BorderLayout.NORTH);
       this.juegoPacman.addObserver(this.panelDibujo);
       this.juegoPacman.addObserver(this.panelInformacion);
       this.juegoPacman.notificar();
   }

   public Juego getJuego(){
```

```
        return this.juegoPacman;
    }

    public static void main(String[] args) {
        FPrincipal frame = new FPrincipal();
        frame.setVisible(true);
    }
}
```

Esta clase crea la instancia de la clase *Juego*. También crea los páneles de información, dibujo y controles. En el constructor se agregan los páneles al *frame* y se agregan observadores a la instancia de la clase *Juego,* que es el modelo para esta aplicación.

Clase *PInformacion*

```
package presentacion;

import java.awt.FlowLayout;
import java.util.Observable;
import java.util.Observer;

import javax.swing.BorderFactory;
import javax.swing.JLabel;
import javax.swing.JPanel;

import logica.Juego;

public class PInformacion extends JPanel implements Observer{
    private JLabel puntos;

    public PInformacion (){
        this.setLayout(new FlowLayout());
        this.setBorder(BorderFactory.createTitledBorder(
        "Informacion"));
        this.puntos = new JLabel("Puntos: ");
        this.add(puntos);
    }

    @Override
    public void update(Observable arg0, Object arg1) {
        this.puntos.setText("Puntos: "+((Juego)arg1).getPuntos());
    }
}
```

En esta clase se actualiza la información de los puntos mediante el método *Update*, una vez que el modelo realice una notificación de cambio de estado.

Clase *PDibujo*

```
package presentacion;

import java.awt.Color;
import java.awt.Graphics;
import java.util.Observable;
import java.util.Observer;

import javax.swing.JPanel;

import logica.Bloque;
import logica.Galleta;
import logica.Juego;

public class PDibujo extends JPanel implements Observer{
   private Object objObservable;

   public void paint(Graphics g){
       Juego juego = (Juego)this.objObservable;
       g.setColor(new Color(20,20,20));
       g.fillRect(0, 0, this.getWidth(), this.getHeight());
       for(Bloque bloque : juego.getBloques()){
          g.drawImage(bloque.getImagen(), bloque.getX(
          )*20, bloque.getY()*20, this);
       }
       for(Galleta galleta : juego.getGalletas()){
       g.drawImage(galleta.getImagen(), galleta.getX(
       )*20, galleta.getY()*20, this);
       }
       g.drawImage(juego.getComeGalletas().getImagen(),
       juego.getComeGalletas().getX()*20, juego.getComeGalletas(
       ).getY()*20, this);
   }

   @Override
   public void update(Observable arg0, Object arg1) {
       this.objObservable = arg1;
       this.repaint();
   }
}
```

En esta clase se actualiza gráficamente el estado del juego mediante el método *Update* una vez que el modelo realice una notificación de cambio de estado.

Clase *PControles*

```
package presentacion;

import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;

import logica.Juego;

public class PControles extends JPanel{
   private Juego juego;
   private JButton arriba;
   private JButton abajo;
   private JButton derecha;
   private JButton izquierda;
   private JLabel espacio1;
   private JLabel espacio2;

   public PControles (Juego juego){
       this.juego=juego;
       this.setLayout(new GridLayout(2,3,0,0));
       this.setBorder(BorderFactory.createTitledBorder(
       "ControlesPacman"));
       {
          this.espacio1 = new JLabel("");
          this.add(this.espacio1);
          this.arriba = new JButton("^");
          this.arriba.addActionListener(new ActionListener() {
              public void actionPerformed(ActionEvent evt) {
                 arribaActionPerformed(evt);
              }
          });
          this.add(arriba);
          this.espacio2 = new JLabel("");
          this.add(this.espacio2);
       }
       {
          this.izquierda = new JButton("<");
          this.izquierda.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent evt) {
               izquierdaActionPerformed(evt);
            }
         });
         this.add(izquierda);
      }
      {
         this.abajo = new JButton("v");
         this.abajo.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
               abajoActionPerformed(evt);
            }
         });
         this.add(abajo);
      }
      {
         this.derecha = new JButton(">");
         this.derecha.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
               derechaActionPerformed(evt);
            }
         });
         this.add(derecha);
      }
   }

   private void arribaActionPerformed(ActionEvent evt) {
       this.juego.arriba();
   }

   private void abajoActionPerformed(ActionEvent evt) {
       this.juego.abajo();
   }

   private void derechaActionPerformed(ActionEvent evt) {
       this.juego.derecha();
   }

   private void izquierdaActionPerformed(ActionEvent evt) {
       this.juego.izquierda();
   }
}
```

En esta clase se crean los controles y los eventos que invocan los métodos que modifican el estado del modelo.

Al ejecutar la aplicación se presenta el juego Pacman.

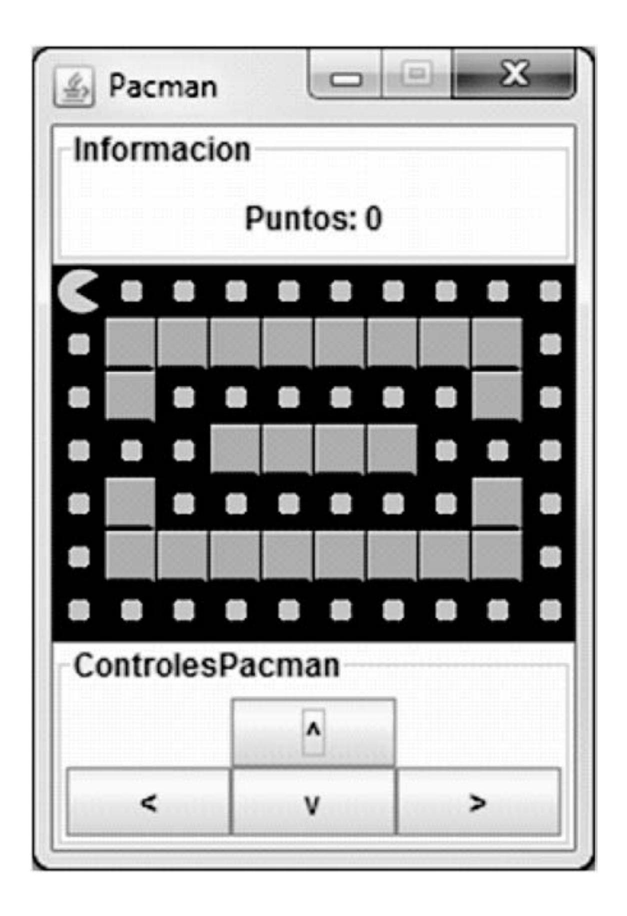


Figura 86. Juego Pacman basado en MVC

Al oprimir dos veces el botón “>”, este control modifica el estado del modelo mediante el método “*derecha*” de la clase *Juego*. Al ejecutarse este método se actualiza el estado de las coordenadas del jugador y se eliminan las galletas correspondientes. Una vez realizado estos procesos se invoca el método “*notificar*”, de la clase *Juego*, el cual invoca los métodos “*setChange*” y “*notifyObservers*” de la súper clase *Observable*.

Las clases *PInformacion* y *PDibujo*, ejecutan el método sobre escrito “*update*”, el cual se encarga de actualizar la vista con base en el modelo modificado. El resultado de estas operaciones es el siguiente:



Figura 87. Juego Pacman basado en MVC

14.3 Ejercicios propuestos

1. Implemente una aplicación utilizando el patrón *MVC* que emule el juego "*puzzle*". Este juego consiste en una cuadrícula de cuatro filas y cuatro columnas, lo que implica tener 16 celdas. Las 15 primeras celdas contienen imágenes que, en el orden correcto, conforman una imagen total. El juego debe entregar las imágenes de las celdas en desorden con el fin de ser ordenadas por el

usuario. El juego debe contener al menos, los siguiente elementos:

a. Modelo. Se debe conformar por una clase *Juego* y una clase *Ficha*. La clase *Ficha* debe tener la posición en el eje *X* y *Y*, además de la imagen. La clase *Juego* debe heredar de *Observable*.

b. Vista. Debe tener un *JPanel* que permita visualizar el estado del juego. Debe implementar la interfaz *Observer*.

c. Controlador. Debe tener un *JPanel* con botones para que el usuario pueda cambiar el estado del modelo.

2. Implemente una aplicación que permita consultar información de productos, de una base de datos, mediante una tabla y un filtro. La aplicación debe contar con al menos los siguientes elementos:

   a. Modelo. El patrón *DAO* para el acceso a los datos debe ser una dependencia de la capa de lógica. De esta manera, las capas, *lógica* y *persistencia* conforman el modelo de la aplicación.

   b. Vista. Debe haber un *JPanel* que contenga un *JTable* que permita visualizar la información capturada de la base de datos.

   c. Controlador. Debe haber un cuadro de texto que tiene el evento *keyReleased*. A través de este evento se modifica el modelo que, consiste en traer los nuevos datos para ser visualizados en el *JTable* de la vista.

CAPÍTULO 15

Procesos multitarea

Los procesadores y sistemas operativos permiten realizar múltiples procesos de forma simultánea.

La máquina virtual Java es un sistema multihilo, es decir, es capaz de ejecutar varios hilos de ejecución simultáneamente. Los hilos se conocen como "*Thread*". Un hilo se puede definir como un único flujo de ejecución dentro de un proceso. La *JVM* puede gestionar asignación de tiempos de ejecución, prioridades, etc.; de forma similar a cómo se gestionan múltiples procesos en un sistema operativo. Sin embargo, los hilos en Java se ejecutan dentro de la *JVM,* siendo este un proceso del sistema operativo, lo cual le obliga a compartir todos los recursos. A este tipo de procesos, donde se comparten los recursos, se les llama procesos ligeros. Entonces, un proceso se compone de un conjunto de hilos o procesos ligeros.

Los hilos son bastante útiles para el desarrollo de *software,* debido a que permiten que el flujo del programa sea dividido en múltiples partes. Cada una de estas partes se debe dedicar de forma independiente a una tarea específica.

Java ofrece soporte para la gestión de hilos mediante las siguientes clases e interfaces:

- *Thread*. La clase *Thread* es la clase responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase se deriva la clase de *Thread* y se sobre escribe el método *run*. El método *run* contiene el código de ejecución de un hilo. La clase *Thread* también define el método *start,* el cual permite iniciar la ejecución del hilo.

- *Runnable*. La interfaz *Runnable* proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando la interfaz, en lugar de derivarla de la clase *Thread*, esto debido a que la clase que requiere la implementación del hilo deba extender de alguna otra clase. Las clases que implementan la interfaz *Runnable* proporcionan un método *run* que debe ser ejecutado por otra instancia.

- *ThreadDeath*. La clase *ThreadDeath* proporciona un mecanismo que permite hacer limpieza, después de que un hilo haya sido finalizado de forma asíncrona. Cuando el método *stop* de un hilo es invocado, una instancia de *ThreadDeath* es lanzada por el hilo.

- *ThreadGroup*. La clase *ThreadGroup* se utiliza para manejar un grupo de hilos. Esta clase es un mecanismo para controlar de modo eficiente la ejecución de un conjunto de hilos. Esta clase proporciona métodos *stop*, *suspend* y *resume*; para controlar la ejecución de todos los hilos pertenecientes al grupo. Los grupos de hilos también pueden contener otros grupos de hilos, permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo pero no al padre del grupo.

- *Object.* La clase *Object* proporciona métodos necesarios dentro de la arquitectura multihilo de Java. Estos métodos son *wait*, *notify* y *notifyAll*. El método *wait* hace que el hilo de ejecución espere en estado dormido hasta que se le notifique que continúe. El método *notify* informa a un hilo en espera de que continúe con su ejecución. El método *notifyAll* es similar a *notify* excepto que se aplica a todos los hilos en espera. Estos tres métodos solo pueden ser llamados desde un método o bloque sincronizado. Normalmente, estos métodos se utilizan cuando hay ejecución multihilo, es decir, cuando un método espera a que otro método termine de hacer algo antes de poder continuar. El primer hilo espera hasta que otro hilo le notifique que puede continuar.

15.1 Creación de hilos

En Java hay dos formas básicas de creación de hilos. La primera forma es mediante el uso de la clase *Threat* y la segunda es mediante el uso de la interfaz *Runnable*.

15.1.1 Creación de hilos mediante la clase *Thread*

En este método es necesario crear una clase que herede de la clase *Thread* y sobrecargar el método *run*.

La sintaxis para crear un hilo con base en la clase *Thread* es la siguiente:

```
public class Hilo extends Thread {

   public void run(){
        ..
   }
}
```

En el método *run* se implementa el código correspondiente a la acción que el hilo debe desarrollar. El método *run* no es invocado directa o explícitamente, debido a que los hilos se inician con el método *start*. La clase *Thread* implementa el método *sleep* que permite detener la ejecución por un tiempo establecido en milisegundos.

La siguiente implementación permite crear tres hilos. El constructor de la clase *HiloThread*, recibe el nombre del hilo y un número de iteraciones que es utilizado en el método *run*, el cual se ejecuta al hacer el llamado al método *start*. Dentro del método *run*, simplemente se calcula la hora exacta en el que ejecuta la iteración y hace el llamado al método *sleep*, con el fin de hacer una pausa en milisegundos de la ejecución del hilo.

```
package hilos;

import java.util.Calendar;
```

```
import java.util.GregorianCalendar;

public class HiloThread extends Thread {
   private int iteraciones;
   private Calendar calendario;

   public HiloThread(String nombre, int iteraciones){
       super(nombre);
       this.iteraciones=iteraciones;
   }

   public void run(){
       System.out.println("Hilo: "+this.getName()+" iniciado");
       for(int i=this.iteraciones; i>=1; i--){
          this.calendario=new GregorianCalendar();
          int hora=this.calendario.get(Calendar.HOUR);
          int minuto=this.calendario.get(Calendar.MINUTE);
          int segundo=this.calendario.get(Calendar.SECOND);
          int milisegundo=this.calendario.get(
          Calendar.MILLISECOND);
          System.out.println("\nIteracion numero "+i+"
          del hilo denominado "+this.getName());
       System.out.println("Hora de ejecucion: "+hora+":"+minuto+
       ":"+segundo+":"+milisegundo);
          try {
              sleep(i*50);
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
       }
   }

   public static void main(String[] args) {
       HiloThread hilo1=new HiloThread("uno", 5);
       HiloThread hilo2=new HiloThread("dos", 3);
       HiloThread hilo3=new HiloThread("tres", 6);
       hilo1.start();
       hilo2.start();
       hilo3.start();
   }
}
```

Salida estándar

```
Hilo: uno iniciado
Hilo: dos iniciado
Hilo: tres iniciado

Iteracion numero 6 del hilo denominado tres
Hora de ejecucion: 1:46:10:671

Iteracion numero 5 del hilo denominado uno
Hora de ejecucion: 1:46:10:671

Iteracion numero 3 del hilo denominado dos
Hora de ejecucion: 1:46:10:671

Iteracion numero 2 del hilo denominado dos
Hora de ejecucion: 1:46:10:812

Iteracion numero 4 del hilo denominado uno
Hora de ejecucion: 1:46:10:921

Iteracion numero 1 del hilo denominado dos
Hora de ejecucion: 1:46:10:921

Iteracion numero 5 del hilo denominado tres
Hora de ejecucion: 1:46:10:968

Iteracion numero 3 del hilo denominado uno
Hora de ejecucion: 1:46:11:109

Iteracion numero 4 del hilo denominado tres
Hora de ejecucion: 1:46:11:218

Iteracion numero 2 del hilo denominado uno
Hora de ejecucion: 1:46:11:265

Iteracion numero 1 del hilo denominado uno
Hora de ejecucion: 1:46:11:375

Iteracion numero 3 del hilo denominado tres
Hora de ejecucion: 1:46:11:421

Iteracion numero 2 del hilo denominado tres
Hora de ejecucion: 1:46:11:562

Iteracion numero 1 del hilo denominado tres
Hora de ejecucion: 1:46:11:671
```

15.1.2 Creación de hilos mediante la interfaz *Runnable*

En este método es necesario crear una clase que implemente la interfaz *Runnable* e implementar el método *run*.

La sintaxis para crear un hilo con base en la interfaz *Runnable* es la siguiente:

```
public class Hilo implements Runnable {

   public void run(){
       ..
   }
}
```

En el método *run* se implementa el código correspondiente a la acción que el hilo debe desarrollar. El método *run* no es invocado directa o explícitamente, debido a que los hilos se inician con el método *start*.

Hay dos desventajas claras de la interfaz *Runnable* con respecto a la clase *Thread*, que consisten en que la interfaz *Runnable* no define un atributo de identificación del hilo y no implementa el método *sleep*.

La siguiente implementación permite crear tres hilos. El constructor de la clase *HiloRunnable,* recibe el nombre del hilo y un número de iteraciones que es utilizado en el método *run*. Dentro del método *run* simplemente se calcula, la hora exacta en el que ejecuta la iteración. El nombre del hilo se deposita en el atributo *Nombre,* definido dentro de la clase. La interfaz *Runnable* no implementa el método *start*, entonces para poder iniciar un hilo que implemente *Runnable*, se debe crear una instancia de la clase *Thread* enviándole en el constructor la instancia de la clase que implementa *Runnable*. A través de la instancia de la clase *Thread* es posible iniciar el hilo, ya que esta si cuenta con el método *start*.

```
package hilos;

import java.util.Calendar;
import java.util.GregorianCalendar;
```

```
public class HiloRunnable implements Runnable {
   private int iteraciones;
   private Calendar calendario;
   private String nombre;

   public HiloRunnable(String nombre, int iteraciones){
       this.nombre=nombre;
       this.iteraciones=iteraciones;
   }

   @Override
   public void run() {
       System.out.println("Hilo: "+this.nombre+" iniciado");
       for(int i=this.iteraciones; i>=1; i--){
          this.calendario=new GregorianCalendar();
          int hora=this.calendario.get(Calendar.HOUR);
          int minuto=this.calendario.get(Calendar.MINUTE);
          int segundo=this.calendario.get(Calendar.SECOND);
          int milisegundo=this.calendario.get(Calendar.MILLISECOND);
          System.out.println("\nIteracion numero
          "+i+" del hilo denominado "+this.nombre);
          System.out.println("Hora de ejecucion:
          "+hora+":"+minuto+":"+segundo+":"+milisegundo);
       }
   }

   public static void main(String[] args) {
       HiloRunnable hilo1=new HiloRunnable("uno", 5);
       Thread t1 = new Thread(hilo1);
       t1.start();
       HiloRunnable hilo2=new HiloRunnable("dos", 3);
       Thread t2 = new Thread(hilo2);
       t2.start();
       HiloRunnable hilo3=new HiloRunnable("tres", 6);
       Thread t3 = new Thread(hilo3);
       t3.start();
   }
}
```

Salida estándar

```
Hilo: uno iniciado
Hilo: dos iniciado
Hilo: tres iniciado

Iteracion numero 5 del hilo denominado uno
Hora de ejecucion: 2:41:58:562

Iteracion numero 4 del hilo denominado uno
Hora de ejecucion: 2:41:58:562

Iteracion numero 3 del hilo denominado uno
Hora de ejecucion: 2:41:58:562

Iteracion numero 2 del hilo denominado uno
Hora de ejecucion: 2:41:58:562

Iteracion numero 3 del hilo denominado dos
Hora de ejecucion: 2:41:58:578

Iteracion numero 2 del hilo denominado dos
Hora de ejecucion: 2:41:58:578

Iteracion numero 1 del hilo denominado dos
Hora de ejecucion: 2:41:58:578

Iteracion numero 1 del hilo denominado uno
Hora de ejecucion: 2:41:58:578

Iteracion numero 6 del hilo denominado tres
Hora de ejecucion: 2:41:58:578

Iteracion numero 5 del hilo denominado tres
Hora de ejecucion: 2:41:58:578

Iteracion numero 4 del hilo denominado tres
Hora de ejecucion: 2:41:58:578

Iteracion numero 3 del hilo denominado tres
Hora de ejecucion: 2:41:58:578

Iteracion numero 2 del hilo denominado tres
Hora de ejecucion: 2:41:58:578

Iteracion numero 1 del hilo denominado tres
Hora de ejecucion: 2:41:58:578
```

15.2 Agrupamiento de hilos

Todo hilo en Java hace parte de un grupo de hilos. Los grupos de hilos proporcionan un mecanismo de agrupamiento de múltiples hilos dentro de un único objeto. Gracias a este agrupamiento es posible manipular un conjunto de hilos, en lugar de manipular cada hilo de forma individual. De esta forma, si se desea iniciar o suspender un conjunto de hilos, basta con hacer el proceso al grupo de hilos que los contiene.

Los grupos de hilos de Java están implementados por la clase *ThreadGroup* que pertenece al paquete **java.lang**. En el momento de crear un hilo, la máquina virtual de Java ubica en tiempo de ejecución dicho hilo por defecto en un grupo de hilos.

Un hilo hace parte de un grupo de hilos de forma permanente, lo cual no permite que un hilo pueda moverse de un grupo a otro una vez que ha sido creado.

La sintaxis para crear un grupo de hilos y para asignar hilos al grupo es la siguiente:

```
ThreadGroup grupo1 = new ThreadGroup("Grupo uno");
HiloThread hilo1 = new HiloThread(grupo1,"uno");
HiloThread hilo2 = new HiloThread(grupo1,"dos");
HiloThread hilo3 = new HiloThread(grupo1,"tres");
```

15.3 Sincronización

Existen casos en los que varios hilos requieren utilizar un mismo recurso. En el momento de acceder a dichos recursos, los hilos deben establecer un orden de acceso para poder hacer los diferentes procesos de forma correcta.

Para asegurar en la aplicación, qué hilos concurrentes no colisionan y operan correctamente con recursos compartidos, se crea un sistema que administre dicha concurrencia.

Las secciones de código de una aplicación, que acceden a un mismo recurso desde dos hilos distintos, se denominan "secciones críticas". Para sincronizar dos o más hilos se hace uso del modificador *synchronized,* en aquellos métodos del recurso con los que puedan producirse situaciones de colisión. De esta forma, Java genera un bloqueo controlado con el recurso sincronizado. La sintaxis para sincronizar un método es la siguiente:

```
public synchronized void metodo() {
   ...
}
```

Existe un ejemplo clásico de sincronización en donde existe un productor y un consumidor. El productor no puede producir un recurso si existe otro. El consumidor no puede consumir un recurso si no existe. Entonces, el productor produce un recurso y en ese mismo momento, el consumidor puede consumirlo. Después que el consumidor lo consume, el productor puede producir uno nuevo.

La siguiente implementación resuelve este caso.

Clase *Recurso*

```
package hilos;

public class Recurso {
   private int dato;
   private boolean hayDato = false;

   public synchronized int get() {
      while (hayDato == false) {
         try {
            // espera a que el productor coloque un valor
            wait();
         } catch (InterruptedException e) { }
      }
      hayDato = false;
      // notificar que el valor ha sido consumido
      notifyAll();
      return dato;
   }
```

```
   public synchronized void put(int valor) {
       while (hayDato == true) {
          try {
              // espera a que se consuma el dato
              wait();
          } catch (InterruptedException e) { }
       }
       dato = valor;
       hayDato = true;
       // notificar que el valor ha sido producido.
       notifyAll();
   }
}
```

La clase recurso tiene dos métodos sincronizados que son *get* y *put*. Estos métodos retornan un dato y asignan un dato, respectivamente. Un atributo denominado *hayDato* le permite esperar al método *get* hasta que exista dicho dato. Una vez lee el dato, envía una notificación. El atributo *hayDato* también, permite al método *put* esperar hasta que el dato no exista. Una vez colocado el nuevo dato, envía una notificación.

Clase *Productor*

```
package hilos;

import java.util.Calendar;
import java.util.GregorianCalendar;

public class Productor extends Thread {
   private Recurso contenedor;

   public Productor (Recurso c) {
       contenedor = c;
   }

   public void run() {
       for (int i = 0; i < 10; i++) {
          contenedor.put(i);
          Calendar calendario=new GregorianCalendar();
          int hora=calendario.get(Calendar.HOUR);
          int minuto=calendario.get(Calendar.MINUTE);
```

```
            int segundo=calendario.get(Calendar.SECOND);
            int milisegundo=calendario.get(Calendar.MILLISECOND);
            System.out.println("Productor. Valor: " + i);
            System.out.println("Hora de ejecucion:
            "+hora+":"+minuto+":"+segundo+ ":"+milisegundo);
            try {
                sleep(800);
            } catch (InterruptedException e) { }
        }
    }
}
```

La clase *productor* hace un proceso iterativo donde invoca el método *put* de la clase *recurso*. Imprime la hora exacta en que hace el proceso y hace una interrupción de 800 milisegundos.

Clase *Consumidor*

```
package hilos;

import java.util.Calendar;
import java.util.GregorianCalendar;

public class Consumidor extends Thread {
   private Recurso contenedor;

   public Consumidor (Recurso c) {
       contenedor= c;
   }

   public void run() {
       int dato = 0;
       while(true){
          dato = contenedor.get();
          Calendar calendario=new GregorianCalendar();
          int hora=calendario.get(Calendar.HOUR);
          int minuto=calendario.get(Calendar.MINUTE);
          int segundo=calendario.get(Calendar.SECOND);
          int milisegundo=calendario.get(Calendar.MILLISECOND);
          System.out.println("Consumidor. Valor: " + dato);
          System.out.println("Hora de ejecucion: "+hora+":"
          +minuto+":"+segundo+":"+milisegundo);
       }
   }
}
```

La clase *consumidor* hace permanentemente un proceso iterativo, donde invoca el método *get* de la clase *recurso*. Imprime la hora exacta en que hace el proceso.

Clase *PruebaSincronizacion*

```
package hilos;

public class PruebaSincronizacion {

   public static void main(String[] args) {
       Recurso c = new Recurso ();
       Productor produce = new Productor (c);
       Consumidor consume = new Consumidor (c);
       produce.start();
       consume.start();
   }
}
```

La clase *prueba* crea un recurso, un producto y un consumidor e invoca el método *start,* de dichos hilos.

Los resultados son los siguientes:

Salida estándar

```
Consumidor. Valor: 0
Hora de ejecucion: 2:5:22:734
Productor. Valor: 0
Hora de ejecucion: 2:5:22:734
Productor. Valor: 1
Hora de ejecucion: 2:5:23:531
Consumidor. Valor: 1
Hora de ejecucion: 2:5:23:531
Productor. Valor: 2
Hora de ejecucion: 2:5:24:328
Consumidor. Valor: 2
Hora de ejecucion: 2:5:24:328
Productor. Valor: 3
Hora de ejecucion: 2:5:25:125
Consumidor. Valor: 3
Hora de ejecucion: 2:5:25:125
Productor. Valor: 4
```

```
Hora de ejecucion: 2:5:25:937
Consumidor. Valor: 4
Hora de ejecucion: 2:5:25:937
Consumidor. Valor: 5
Hora de ejecucion: 2:5:26:734
Productor. Valor: 5
Hora de ejecucion: 2:5:26:734
Productor. Valor: 6
Hora de ejecucion: 2:5:27:531
Consumidor. Valor: 6
Hora de ejecucion: 2:5:27:531
Consumidor. Valor: 7
Hora de ejecucion: 2:5:28:328
Productor. Valor: 7
Hora de ejecucion: 2:5:28:328
Productor. Valor: 8
Hora de ejecucion: 2:5:29:140
Consumidor. Valor: 8
Hora de ejecucion: 2:5:29:140
Productor. Valor: 9
Hora de ejecucion: 2:5:29:937
Consumidor. Valor: 9
Hora de ejecucion: 2:5:29:937
```

15.4 Temporizadores

Un temporizador equivale a la ejecución de un hilo, el cual ejecuta el método *run* en un intervalo de tiempo proporcionado.

La creación de un temporizador se hace mediante la creación de un objeto instancia de la clase *Timer*, la cual pertenece al paquete **java.util**. Al objeto de la clase *Timer* se le asigna un objeto instancia de la clase *TimerTask*, la cual pertenece al paquete **java.util**; con un tiempo de inicio y un tiempo de intervalo medido en milisegundos. Al instanciar la clase *TimerTask* se puede implementar el método *run*, que proporciona la acción que se va a ejecutar.

Esta implementación es bastante simple y útil ,porque no requiere crear un hilo mediante la herencia de *Thread*, ni mediante la implementación de *Runnable*.

La siguiente implementación crea un reloj a través de un *Timer*, al cual se le asigna un *TimerTask* con un intervalo de tiempo de 1000 milisegundos.

```
package hilos;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Timer;
import java.util.TimerTask;

public class Temporizador {
   private Timer timer;

   public static void main(String[] args) {
       Temporizador t=new Temporizador();
       t.iniciarTimer();
   }

   public void iniciarTimer(){
       this.timer=new Timer();
       TimerTask timerTask = new TimerTask(){
          public void run(){
              accion();
          }
       };
       timer.scheduleAtFixedRate(timerTask, 0, 1000);

   }

   private void accion(){
       Calendar calendario=new GregorianCalendar();
       int hora=calendario.get(Calendar.HOUR);
       int minuto=calendario.get(Calendar.MINUTE);
       int segundo=calendario.get(Calendar.SECOND);
       System.out.println("Hora: "+hora+":"+minuto+":"+segundo);
   }
}
```

Los resultados son los siguientes:

Salida estándar

`Hora: 2:24:25`
`Hora: 2:24:26`
`Hora: 2:24:27`
`Hora: 2:24:28`
`Hora: 2:24:29`
`Hora: 2:24:30`
`Hora: 2:24:31`
`Hora: 2:24:32`
`Hora: 2:24:33`
`Hora: 2:24:34`
`Hora: 2:24:35`
`Hora: 2:24:36`
`Hora: 2:24:37`
`Hora: 2:24:38`
`Hora: 2:24:39`
`Hora: 2:24:40`

15.5 Ejercicios propuestos

1. Implemente un hilo que permita hacer un llamado al método *Paint* de un *JPanel*, en un intervalo de 100 milisegundos para pintar un círculo que cambia su posición diagonalmente.

2. Implemente un cronómetro, a través de un *JFrame*, el cual pueda almacenar tiempos parciales en una lista.

3. Amplié el algoritmo de *productor* y *consumidor*, para dos recursos con un productor y un consumidor.

4. Amplié el algoritmo de *productor* y *consumidor*, para un recurso con dos productores y un consumidor.

CAPÍTULO 16

Comunicaciones en red

Java permite realizar aplicaciones que posean comunicación en red mediante el uso del modelo, Cliente Servidor.

16.1 Modelo Cliente Servidor

El modelo *Cliente Servidor* permite la comunicación entre un equipo denominado servidor y un conjunto de equipos denominados clientes.

Los clientes se encuentran en constante comunicación con el servidor y a través de este, pueden establecer comunicación con otros clientes.

El servidor es el encargado de aceptar solicitudes de conexión por parte de los clientes y los clientes deben conectarse al servidor a través de su dirección lógica y de su puerto de aplicación. Esto significa que todas las gestiones que se realizan se concentran en el servidor, de manera que en él se disponen los diferentes requerimientos del sistema.

16.2 *Socket* y *ServerSocket*

El *Socket* es el elemento que permite establecer una comunicación entre un servidor y múltiples clientes. Es posible trabajar comunicaciones mediante la clase *Socket* del paquete **java.net**. Esta clase permite crear conexiones que utiliza el protocolo *TCP*, entre dos computadores. Los *sockets* son conectores entre dos computadores

remotos, en donde su comunicación es continua y finaliza cuando uno de los dos computadores cierra su conexión. La clase *ServerSocket* es una clase que pertenece al paquete **java.net**, que sirve para atender peticiones de conexiones, lo cual es necesario para la creación de un servidor.

Para crear un servidor se debe crear un *socket* y un *ServerSocket*. El *ServerSocket* debe tener como parámetro el número de puerto de la aplicación. Este número de puerto debe ser superior a 1024, con el fin de que la aplicación no presente conflicto con otras aplicaciones o protocolos de redes de computadores.

La clase *ServerSocket* tiene el método *accept*, el cual permite aceptar una conexión. Esto indica que el servidor debe tener un proceso iterativo que tenga un proceso para aceptar conexiones. El método *accept* retorna un *socket* que contiene la información del cliente remoto que se ha conectado.

La clase *Socket* posee los métodos *getInputStream* que retorna un *Stream* con el cual, el servidor puede recibir información por parte del cliente conectado y *getOutputStream*, que retorna un *Stream* con el cual el servidor puede enviar información al cliente conectado.

La implementación para crear un servidor, con una sola conexión por parte de un cliente, es la siguiente:

```
try {
   ServerSocket serverSocket = new ServerSocket(5000);
   System.out.println("Esperando nueva conexion");
   Socket socket = serverSocket.accept();
   System.out.println("Conexion desde:
   "+socket.getInetAddress().getHostName()+" IP:
   "+socket.getInetAddress().getHostAddress());
   ObjectInputStream input = new ObjectInputStream(
   socket.getInputStream());
   ObjectOutputStream output = new
   ObjectOutputStream(socket.getOutputStream());
} catch (IOException e) {
   e.printStackTrace();
}catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

En esta implementación de servidor se obtiene un *socket* al presentarse una conexión. La comunicación se establece a través del puerto 5000. La clase *Socket,* además provee el método *getInetAddress*, el cual retorna un *object* instancia de la clase *InetAddress* que a su vez cuenta con los métodos *getHostName* y *getHostAddress*, que permiten capturar el nombre y dirección *IP* respectivamente, del cliente conectado.

El cliente puede conectarse al servidor, una vez que este se encuentre esperando conexión. Entonces, en el cliente debe crearse un *socket* al cual se le envía por parámetro una dirección *IP* a través de la clase *InetAddress* y el número del puerto de servidor. Si el cliente se encuentra en la misma máquina del servidor, se puede hacer uso de la dirección *localhost* o 127.0.0.1.

```
try {
   Socket socket = new Socket(
   InetAddress getByName(127.0.0.1),5000);
   ObjectInputStream input = new ObjectInputStream(
   socket.getInputStream());
   ObjectOutputStream output = new ObjectOutputStream(
   socket.getOutputStream());
} catch (IOException e) {
   e.printStackTrace();
}catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

De la misma forma que en el servidor se puede obtener un *Stream* para enviar datos y otro para recibir datos. El modelo cliente servidor indica que se puede conectar un servidor con múltiples clientes. Para ello, es necesario modificar la implementación anterior con el fin de permitir múltiples conexiones.

La implementación para crear un servidor con múltiples conexiones por parte de clientes es la siguiente:

```
try {
   ServerSocket serverSocket = new ServerSocket(5000);
   int contador=1;
   while(true){
       System.out.println("Esperando nueva conexion");
```

```
        Socket socket = serverSocket.accept();
        System.out.println( "Conexion " + contador +" desde:" +
        socket.getInetAddress().getHostName()+" IP:"+
        socket.getInetAddress().getHostAddress());
        contador++;
        ObjectInputStream input = new ObjectInputStream(
        socket.getInputStream());
        ObjectOutputStream output = new
        ObjectOutputStream(socket.getOutputStream());
   }
} catch (IOException e) {
   e.printStackTrace();
}catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

Esta implementación atiende las conexiones de clientes mediante un proceso iterativo sin fin. Sin embargo, en el momento de recibir una conexión, crea los *Stream* de entrada y salida, pero ejecuta la siguiente iteración y queda de nuevo esperando conexión. Esto indica que este servidor estará en capacidad, únicamente, de aceptar conexiones, pero no podrá enviar ni recibir mensajes.

Para agregar funcionalidad de envió y recepción de mensajes es necesario implementar un hilo, para que el servidor cree una instancia del hilo por cada conexión. Entonces, a través de cada hilo, el servidor puede enviar y recibir información. La implementación del hilo es la siguiente:

```
package servidor;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class HiloServidor extends Thread {
   private ObjectOutputStream output;
   private ObjectInputStream input;
   private Servidor servidor;
   boolean activo=true;
   private String mensajeRecibido;

   public HiloServidor(String nombre, ObjectInputStream input,
   ObjectOutputStream output, Servidor servidor){
       super(nombre);
       this.input=input;
```

```
        this.output=output;
        this.servidor=servidor;
    }

    public void run(){
        while(this.activo){
            try {
                this.mensajeRecibido = (
                String)this.input.readObject();
                System.out.println(this.getName()+" dice: "+
                this.mensajeRecibido);
                this.servidor.enviarMensaje(this.mensajeRecibido);
            } catch (IOException e) {
                this.activo=false;
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }

    public void enviar (String mensaje){
        try {
            this.output.writeObject(mensaje);
            this.output.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Este hilo recibe a través del constructor, un nombre del hilo, un *Stream* de entrada, un *Stream* de salida y una referencia al servidor. Esta última, se crea con el objetivo de tener acceso a los servicios implementados en el servidor. El método *run* se encarga de recibir información a través del *Stream* de entrada, por medio del método *readObject*, al cual se le debe aplicar un *casting*. El método *enviar* se encarga de enviar información a través del *Stream* de salida, por medio de los métodos *writeObject* y *flush*.

Para que el servidor pueda hacer uso del hilo debe modificarse de tal forma que, por cada conexión pueda crear un hilo y almacenarlo. Al crear un hilo, el servidor debe enviar el nombre del hilo, el *Stream* de entrada, el *Stream* de salida y el servidor, a través de la sentencia *this*. La implementación del nuevo servidor es la siguiente:

```
package servidor;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Iterator;

public class Servidor {
   private Socket socket;
   private ServerSocket serverSocket;

   public void iniciarServidor(){
       try {
          this.serverSocket = new ServerSocket(5000);
          int contador=1;
          while(true){
              System.out.println("Esperando nueva conexion");
              this.socket = serverSocket.accept();
                 System.out.println("Conexion " + contador +
                 " desde: " +
                 this.socket.getInetAddress().getHostName()+" IP: "+
                 this.socket.getInetAddress().getHostAddress());
              contador++;
              ObjectInputStream input = new
              ObjectInputStream(this.socket.getInputStream());
              ObjectOutputStream output = new
              ObjectOutputStream(this.socket.getOutputStream());
              HiloServidor hiloServidor = new
              HiloServidor("Cliente "+contador, input, output,this);
              hiloServidor.start();
          }
       } catch (IOException e) {
          e.printStackTrace();
       }catch (ClassNotFoundException e) {
          e.printStackTrace();
       }
   }

   public static void main(String[] args) {
       Servidor servidor = new Servidor();
       servidor.iniciarServidor();
   }
}
```

El método *iniciarServidor* se encarga de recibir conexiones. Por cada conexión recibida, crea un hilo que adiciona en un vector. Cada hilo

creado envía por el constructor el nombre, el *Stream* de entrada, el *Stream* de salida y una referencia al servidor.

El método *enviarMensaje* envía un mensaje que recibe por parámetro a todos los clientes, mediante un proceso iterativo de los hilos almacenados en el vector.

En la aplicación del cliente, también se requiere crear un hilo que se encarga de recibir mensajes por parte del servidor. La implementación es la siguiente:

```
package cliente;

import java.io.IOException;
import java.io.ObjectInputStream;

public class HiloCliente extends Thread {
   private ObjectInputStream input;
   private boolean activo=true;

   public HiloCliente(ObjectInputStream input) {
       this.input=input;
   }

   public void run(){
       while(this.activo){
          String mensaje;
          try {
              mensaje = (String)this.input.readObject();
              System.out.println(mensaje);
          } catch (IOException e) {
              this.activo=false;
          } catch (ClassNotFoundException e) {
              e.printStackTrace();
          }
       }
   }
}
```

El método *run* de forma permanente intenta leer un mensaje por parte del servidor. En caso que ocurra una excepción, el atributo activo pasa a falso y el hilo deja de leer a través del *Stream* de entrada que fue recibido por el constructor.

16.3 *Chat*

Una de las aplicaciones clásicas del modelo Cliente Servidor, mediante *sockets*, es el *chat*. Un *chat* debe tener un servidor que reciba conexiones de múltiples clientes. Cuando un cliente se conecta, el servidor notifica a todos los clientes que este nuevo cliente se ha conectado. Cuando un cliente envía un mensaje, el servidor reenvía este mensaje a todos los clientes.

La implementación del *chat* es la siguiente:

Clase *Servidor*

```
package servidor;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;

public class Servidor {
   private Socket socket;
   private ServerSocket serverSocket;
   private String nombreUsuarioConectado;
   private ArrayList<HiloServidor> hilosServidor = new
   ArrayList<HiloServidor>();

   public void iniciarServidor(){
       try {
          this.serverSocket = new ServerSocket(5000);
          int contador=1;
          while(true){
              System.out.println("Esperando nueva conexion");
              this.socket = serverSocket.accept();
                 System.out.println( "Conexion "
                 + contador +" desde: " +
                 this.socket.getInetAddress().getHostName()+" IP: "+
                 this.socket.getInetAddress().getHostAddress());
              contador++;
              ObjectInputStream input = new
              ObjectInputStream(this.socket.getInputStream());
              ObjectOutputStream output = new
              ObjectOutputStream(this.socket.getOutputStream());
```

```
                this.nombreUsuarioConectado = (
                String)input.readObject();
                System.out.println(this.nombreUsuarioConectado+
                " se ha conectado");
                HiloServidor hiloServidor = new
                HiloServidor(this.nombreUsuarioConectado,
                input, output, this);
                hiloServidor.start();
                this.hilosServidor.add(hiloServidor);
                this.enviarClienteConectado();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void enviarClienteConectado(){
        for(HiloServidor hilo : this.hilosServidor){
            hilo.enviar(this.nombreUsuarioConectado+
            " se ha conectado");
        }
    }

    public void enviarMensaje(String mensaje){
        for(HiloServidor hilo : this.hilosServidor){
            hilo.enviar(mensaje);
        }
    }

    public static void main(String[] args) {
        Servidor servidor = new Servidor();
        servidor.iniciarServidor();
    }
}
```

En la clase *Servidor*, el método *iniciarServidor* se encarga de recibir conexiones por parte de clientes. Una vez hecha la conexión recibe un nombre de cliente, el cual será el identificador del hilo.

El método *enviarClienteConectado* se encarga de enviar a todos los clientes una notificación de conexión de un nuevo cliente.

El método *enviarMensaje* se encarga de enviar el mensaje que recibe por parámetro a todos los clientes conectados.

Clase *HiloServidor*

```
package servidor;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class HiloServidor extends Thread {
   private ObjectOutputStream output;
   private ObjectInputStream input;
   private Servidor servidor;
   boolean activo=true;
   private String mensajeRecibido;

   public HiloServidor(String nombre, ObjectInputStream input,
   ObjectOutputStream output, Servidor servidor){
       super(nombre);
       this.input=input;
       this.output=output;
       this.servidor=servidor;
   }

   public void run(){
       while(this.activo){
          try {
              this.mensajeRecibido = (
              String)this.input.readObject();
              System.out.println(this.getName()+" dice: "+
              this.mensajeRecibido);
              this.servidor.enviarMensaje(this.mensajeRecibido);
          } catch (IOException e) {
              this.activo=false;
          } catch (ClassNotFoundException e) {
              e.printStackTrace();
          }
       }
   }

   public void enviar (String mensaje){
       try {
          this.output.writeObject(mensaje);
          this.output.flush();
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

En la clase *HiloServidor*, el método *run* se encarga de recibir mensajes del cliente asociado con el hilo. Una vez recibido un mensaje, invoca el método *enviarMensaje* de la clase *Servidor* con el fin de enviar dicho mensaje recibido a todos los clientes.

El método *enviar* se encarga de enviar al cliente asociado con el hilo, el mensaje recibido por parámetro.

Clase *FCliente*

```
package cliente;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.WindowConstants;

public class FCliente extends javax.swing.JFrame {
   private JPanel panelConexion;
   private JTextField textIP;
   private JTextField textNombre;
   private JButton buttonEnviar;
   private JTextField textMensaje;
   private JButton buttonConectar;
   private JLabel labelNombre;
   private JLabel labelIP;
   private JTextArea textMensajes;
   private JPanel panelMensaje;
   private Cliente cliente;
   private Socket socket;
   private HiloCliente hiloCliente;
   private ObjectOutputStream output;
   private ObjectInputStream input;
```

```
public static void main(String[] args) {
    FCliente frame = new FCliente();
    frame.setVisible(true);
}

public FCliente() {
    initGUI();
    this.cliente = new Cliente(this);
}

public JTextArea getTextMensajes(){
    return this.textMensajes;
}

private void initGUI() {
    setDefaultCloseOperation(
    WindowConstants.DISPOSE_ON_CLOSE);
    setTitle("Mini Chat. Cliente");
    getContentPane().setLayout(new BorderLayout());
    {
       panelConexion = new JPanel();
       panelConexion.setLayout(new FlowLayout());
       getContentPane().add(
       panelConexion, BorderLayout.NORTH);
       {
           labelIP = new JLabel();
           panelConexion.add(labelIP);
           labelIP.setText("IP:");
       }
       {
           textIP = new JTextField();
           panelConexion.add(textIP);
           textIP.setText("127.0.0.1");
       }
       {
           labelNombre = new JLabel();
           panelConexion.add(labelNombre);
           labelNombre.setText("Nombre Cliente");
       }
       {
           textNombre = new JTextField();
           panelConexion.add(textNombre);
           textNombre.setPreferredSize(
           new java.awt.Dimension(100, 20));
           textNombre.setSize(80, 20);
       }
       {
           buttonConectar = new JButton();
```

```
            panelConexion.add(buttonConectar);
            buttonConectar.setText("Conectar");
            buttonConectar.addActionListener(
            new ActionListener() {
               public void actionPerformed(ActionEvent evt) {
                   BConectarActionPerformed(evt);
               }
            });
        }
    }
    {
        panelMensaje = new JPanel();
        BorderLayout PMensajeLayout = new BorderLayout();
        panelMensaje.setLayout(PMensajeLayout);
        getContentPane().add(
        panelMensaje, BorderLayout.SOUTH);
        {
            textMensaje = new JTextField();
            panelMensaje.add(textMensaje, BorderLayout.CENTER);
            textMensaje.setEnabled(false);
            textMensaje.setPreferredSize(
            new java.awt.Dimension(348, 21));
        }
        {
            buttonEnviar = new JButton();
            panelMensaje.add(buttonEnviar, BorderLayout.EAST);
            buttonEnviar.setText("Enviar");
            buttonEnviar.addActionListener(
            new ActionListener() {
               public void actionPerformed(ActionEvent evt) {
                   buttonEnviarActionPerformed(evt);
               }
            });
            buttonEnviar.setEnabled(false);
        }
    }
    {
        textMensajes = new JTextArea();
        getContentPane().add(
        textMensajes, BorderLayout.CENTER);
    }
    setSize(400, 300);
}

private void BConectarActionPerformed(ActionEvent evt) {
    if(!this.textNombre.getText().equals("")){

        if(this.cliente.conectar(this.textIP.getText(
```

```
            ),this.textNombre.getText())){
                this.buttonConectar.setEnabled(false);
                this.textNombre.setEnabled(false);
                this.textIP.setEnabled(false);
                this.textMensaje.setEnabled(true);
                this.buttonEnviar.setEnabled(true);
            }else{
                JOptionPane.showMessageDialog(this,
                "Error de servidor","Error",
                JOptionPane.ERROR_MESSAGE);
            }
        }else{
            JOptionPane.showMessageDialog(this,
            "Inserte Nombre","Error",JOptionPane.ERROR_MESSAGE);
        }
    }

    private void buttonEnviarActionPerformed(ActionEvent evt) {
        this.cliente.enviar(this.textMensaje.getText());
        this.textMensaje.setText("");
    }
}
```

La clase *FCliente* es un *JFrame* que ejecuta el cliente, para enviar y recibir mensajes del servidor. Este *frame* contiene un cuadro de texto para ingresar la dirección *IP* donde se ubica el servidor, un cuadro de texto para ingresar el nombre del cliente, un botón *Conectar* para realizar la conexión con el servidor, un cuadro de texto para que el cliente digite el mensaje que desea enviar, un botón que envía el mensaje del cuadro de texto anterior y un área de texto que permite incluir los mensajes recibidos.

El método *buttonConectarActionPerformed* se invoca al hacer clic en el botón *conectar* y efectúa la conexión con el servidor a través de un *socket* indicando la dirección *IP* del servidor y puerto de la aplicación. Una vez el servidor atiende la conexión se crean el *Stream* de entrada, el *Stream* de salida y el hilo del cliente.

El método *buttonEnviarActionPerformed* envía información al servidor a través del *Stream* de salida obtenido por el *socket* en el cliente. Estas funcionalidades se implementan en las siguientes clases:

Clase *Cliente*

```
package cliente;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;

public class Cliente {
   private FCliente frameCliente;
   private Socket socket;
   private HiloCliente hiloCliente;
   private ObjectOutputStream output;
   private ObjectInputStream input;

   public Cliente(FCliente frameCliente) {
       this.frameCliente = frameCliente;
   }

   public boolean conectar(String ip, String nombre){
       try {
          this.socket = new Socket(
          InetAddress.getByName(ip),5000);
          this.output = new ObjectOutputStream(
          this.socket.getOutputStream());
          this.input = new ObjectInputStream(
          this.socket.getInputStream());
          this.output.writeObject(nombre);
          this.output.flush();
          this.hiloCliente = new HiloCliente(
          this.frameCliente,this.input);
          this.hiloCliente.start();
       } catch (Exception e) {
          return false;
       }
       return true;
   }

   public void enviar(String mensaje){
       try {
          this.output.writeObject(mensaje);
          this.output.flush();
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

Clase *HiloCliente*

```
package cliente;

import java.io.IOException;
import java.io.ObjectInputStream;

public class HiloCliente extends Thread {
   private ObjectInputStream input;
   private FCliente cliente;
   private boolean activo=true;

   public HiloCliente(FCliente cliente,ObjectInputStream input) {
       this.cliente=cliente;
       this.input=input;
   }

   public void run(){
       while(this.activo){
          String mensaje;
          try {
              mensaje = (String)this.input.readObject();
              this.cliente.getTMensajes().append(mensaje+"\n");
          } catch (IOException e) {
              this.activo=false;
          } catch (ClassNotFoundException e) {
              e.printStackTrace();
          }
       }
   }
}
```

En la clase *HiloCliente*, el método *run* se encarga de recibir mensajes del servidor. Una vez recibido un mensaje, lo coloca en el área de texto del *frame* del cliente.

16.3.1 Prueba de *chat*

Al ejecutar el servidor queda en espera de una conexión. Es importante anotar que el servidor se encuentra en la misma máquina del cliente,

por lo que se usa la dirección *IP* 127.0.0.1. El servidor presenta la siguiente información en consola:



Salida estándar
Esperando nueva conexión

Al ejecutar un primer cliente se presenta el *frame* de la figura anterior. Al ingresar un nombre de cliente y dar clic en el botón *Conectar*, se realiza la conexión de este cliente con el servidor y se presenta la notificación de conexión. En este caso, en el cliente uno se presenta el mensaje "*Cliente uno se ha conectado*". El resultado se presenta en la Figura 88.



Figura 88. Conexión cliente uno de *chat*

En el servidor se captura la información de dicho cliente. Se presenta la siguiente información en consola:

Salida estándar
Esperando nueva conexión Conexión 1 desde: localhost IP: 127.0.0.1 Cliente uno se ha conectado Esperando nueva conexión

Al ejecutar el segundo cliente, ingresar un nombre de cliente y dar clic en el botón *Conectar*, se realiza la conexión de este cliente con el servidor y se presenta la notificación de conexión. En este caso, tanto en el cliente uno como en el cliente dos, se presenta el mensaje "*Cliente dos se ha conectado*". El resultado se presenta en la Figura 89.



Figura 89. Conexión cliente dos de *chat*

En el servidor se captura la información de dicho cliente. Se presenta la siguiente información en consola:

Salida estándar

```
Esperando nueva conexión
Conexión 1 desde: localhost IP: 127.0.0.1
Cliente uno se ha conectado
Esperando nueva conexión
Conexión 2 desde: localhost IP: 127.0.0.1
Cliente dos se ha conectado
Esperando nueva conexión
```

Al ejecutar el tercer cliente, ingresar un nombre de cliente y dar clic en el botón *Conectar*, se realiza la conexión de este cliente con el servidor y se presenta la notificación de conexión. En este caso, tanto en el cliente uno, como en el cliente dos, como en el cliente tres, se presenta el mensaje "*Cliente tres se ha conectado*". El resultado se presenta en la Figura 90.

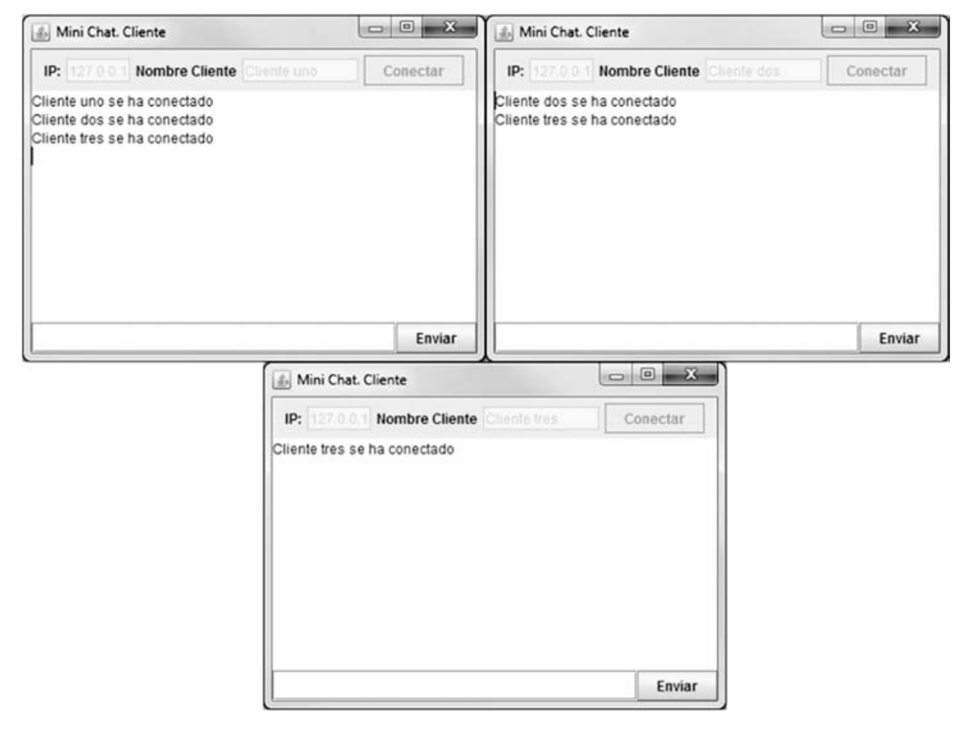


Figura 90. Conexión cliente tres de *chat*

En el servidor se captura la información de dicho cliente. Se presenta la siguiente información en consola.

Salida estándar
Esperando nueva conexión **Conexión 1 desde: localhost IP: 127.0.0.1** **Cliente uno se ha conectado** **Esperando nueva conexión** **Conexión 2 desde: localhost IP: 127.0.0.1** **Cliente dos se ha conectado** **Esperando nueva conexión** **Conexión 3 desde: localhost IP: 127.0.0.1** **Cliente tres se ha conectado** **Esperando nueva conexión**

Al colocar un mensaje en el cuadro de texto inferior del cliente uno y dar clic en enviar, el servidor recibe dicho mensaje y lo reenvía

a todos los clientes conectados. En el mensaje, el servidor agregar el texto "*Cliente dice:*", donde cliente hace referencia al nombre del cliente que envía el mensaje.

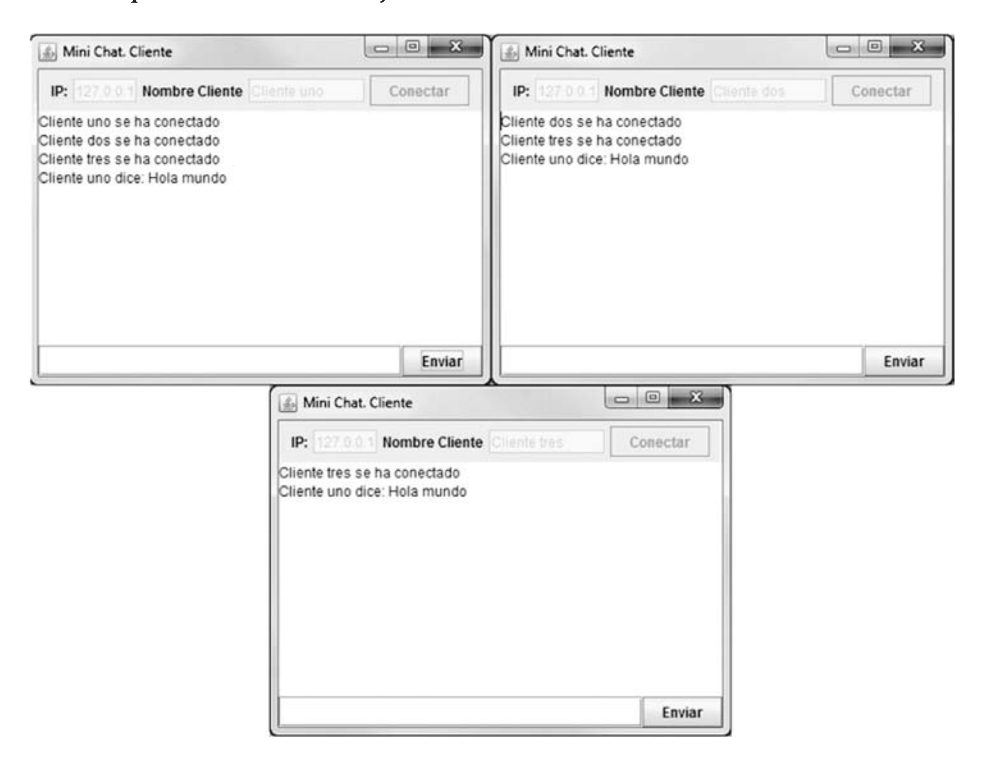


Figura 91. Envío de mensaje de cliente uno de chat

En el servidor se captura el mensaje enviado por el cliente uno, se visualiza en consola y se reenvía a todos los demás clientes. Se presenta la siguiente información en consola:

Salida estándar
Esperando nueva conexión **Conexión 1 desde: localhost IP: 127.0.0.1** **Cliente uno se ha conectado** **Esperando nueva conexión** **Conexión 2 desde: localhost IP: 127.0.0.1** **Cliente dos se ha conectado** **Esperando nueva conexión**

```
Conexión 3 desde: localhost IP: 127.0.0.1
Cliente tres se ha conectado
Esperando nueva conexión
Cliente uno dice: Hola mundo
```

Al colocar un mensaje en el cuadro de texto inferior del cliente dos y dar clic en enviar, el servidor recibe dicho mensaje y lo reenvía a todos los clientes conectados de la misma forma que se realizó el proceso con el cliente uno.

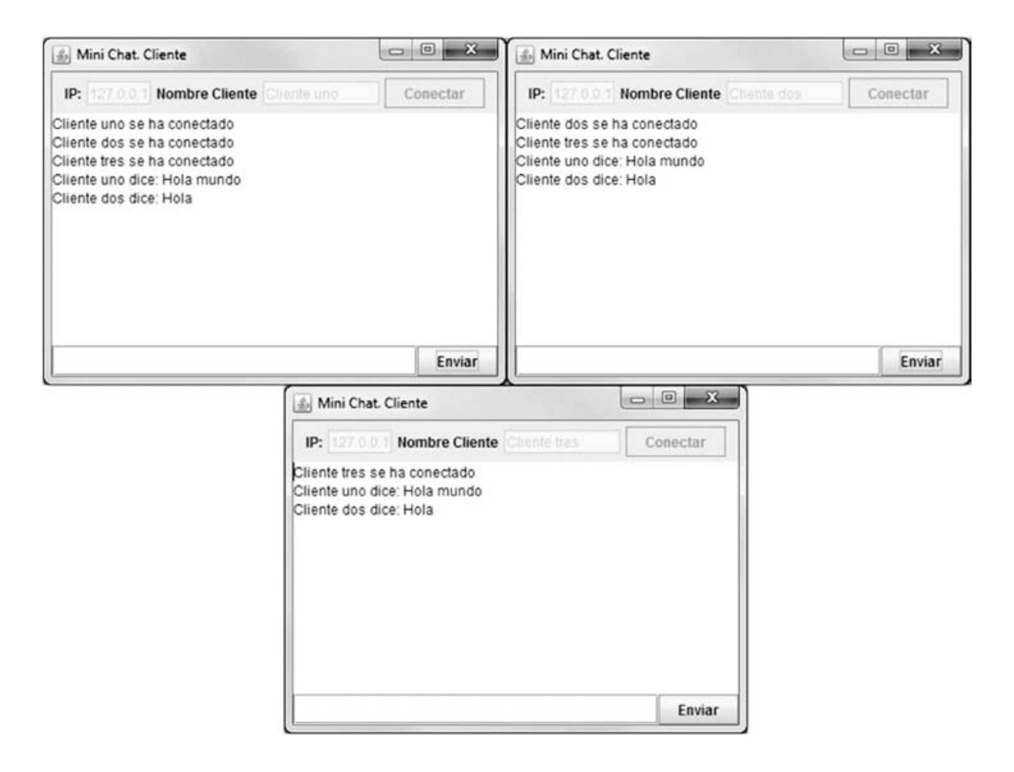


Figura 92. Envío de mensaje de cliente dos de *chat*

En el servidor se captura el mensaje enviado por el cliente uno, se visualiza en consola y se reenvía a todos los demás clientes. Se presenta la siguiente información en consola:

Salida estándar

`Esperando nueva conexión`
`Conexión 1 desde: localhost IP: 127.0.0.1`
`Cliente uno se ha conectado`
`Esperando nueva conexión`
`Conexión 2 desde: localhost IP: 127.0.0.1`
`Cliente dos se ha conectado`
`Esperando nueva conexion`
`Conexión 3 desde: localhost IP: 127.0.0.1`
`Cliente tres se ha conectado`
`Esperando nueva Conexión`
`Cliente uno dice: Hola mundo`
`Cliente dos dice: Hola`

16.4 Ejercicios propuestos

1. Implemente una aplicación cliente servidor que permita emular un *chat* con envío de mensajes privados, es decir, se debe crear una lista de clientes conectados y se debe seleccionar un cliente de la lista para que sea el único destinatario.

2. Implemente una aplicación cliente servidor que permita enviar archivos.

3. Implemente el juego *triqui* en una aplicación cliente servidor que permita la conexión de dos clientes y visualice los resultados de las jugadas de cada uno de ellos.

CAPÍTULO 17

Multimedia

Java permite trabajo con multimedia mediante *API* especializadas. Un *API* bastante conocido es *JMF* (*Java Media Framework*), el cual se debe descargar e instalar. Una vez instalado se agrega al *JDK* (*Java Development Kit*) y *JRE* (*Java Runtime Enviroment*).

El *API JMF* agrega el paquete **javax.media**, el cual contiene clases como *Player*, *CaptureDeviceInfo* y *MediaLocator*, entre otras.

El *JMF* proporciona una arquitectura unificada y un protocolo de mensajes para gestionar la adquisición, procesamiento y entrega de datos multimedia. Soporta la mayoría de las representaciones multimedia como *WAV*, *MPEG*, *AU*, *AVI*, etc. Las aplicaciones *JMF* pueden presentar, capturar, manipular y almacenar información multimedia.

Los datos multimedia pueden obtenerse de diferentes fuentes como archivos locales o de red, cámaras, micrófonos, etc.

El audio y el vídeo se presentan a través de dispositivos de salida como altavoces y monitores. Los flujos multimedia pueden ser enviados a otros destinos, almacenarlos en archivos o transmitirlos a través de la red.

17.1 Captura de vídeo

Una de las aplicaciones más frecuentes es la captura de vídeo a través de este *API*. Para ello se debe implementar el siguiente código:

```
String Webcam = "vfw:Microsoft WDM Image Capture (Win32):0";
CaptureDeviceInfo device = CaptureDeviceManager.
getDevice(Webcam);
MediaLocator mediaLocator = this.device.getLocator();
Player player = Manager.createRealizedPlayer(mediaLocator);
player.start();
Component component = this.player.getVisualComponent();
JPanel PVideo = new JPanel();
PVideo.add(component);
```

Con el código anterior se configura un dispositivo equivalente a la *WebCam*, instalada en el equipo a través del objeto *device*. El objeto *mediaLocator* permite la creación de un objeto *player* que es el encargado de iniciar la cámara *web* para presentar la imagen a través de un *JPanel*.

La siguiente implementación permite colocar un panel con vídeo en un *frame*, con base en clases independientes para la presentación y la captura de vídeo.

Clase *FVideo*

```
package Presentacion;

import java.awt.BorderLayout;
import javax.swing.JPanel;
import javax.swing.WindowConstants;
import javax.swing.SwingUtilities;
import Logica.Video;

public class FVideo extends javax.swing.JFrame {
   private JPanel PVideo;
   private final Video video=new Video();

   public static void main(String[] args) {
       SwingUtilities.invokeLater(new Runnable() {
          public void run() {
              FVideo inst = new FVideo();
              inst.setLocationRelativeTo(null);
```

```
                inst.setVisible(true);
            }
        });
    }

    public FVideo() {
        super();
        initGUI();
        this.iniciarCaptura();
    }

    private void initGUI() {
        try {
            setDefaultCloseOperation(
            WindowConstants.DISPOSE_ON_CLOSE);
            this.setTitle("Captura de Video");
            {
                PVideo = new JPanel();
                BorderLayout PVideoLayout = new BorderLayout();
                PVideo.setLayout(PVideoLayout);
                getContentPane().add(PVideo, BorderLayout.CENTER);
            }
            pack();
            setSize(400, 300);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void iniciarCaptura(){
        this.video.capturar();
        this.PVideo.add(this.video.getComponent());
    }
}
```

Clase *Video*

```
package Logica;

import java.awt.Component;
```

```
import javax.media.CaptureDeviceInfo;
import javax.media.CaptureDeviceManager;
import javax.media.Manager;
import javax.media.MediaLocator;
import javax.media.Player;

public class Video {
   private Player player = null;
   private CaptureDeviceInfo device = null;
   private MediaLocator mediaLocator = null;
   private Component component=null;

   public Component getComponent() {
       return component;
   }

   public void capturar(){
       String Webcam = "vfw:Microsoft WDM Image Capture (
       Win32):0";
       this.device = CaptureDeviceManager.getDevice(Webcam);
       this.mediaLocator = this.device.getLocator();
       try  {
          this.player = Manager.createRealizedPlayer(
          this.mediaLocator);
          this.player.start();
          this.component = this.player.getVisualComponent();
       } catch (Exception e)  {
          e.printStackTrace();
       }
   }

   public void detener(){
       try  {
          this.player.close();
       } catch (Exception e)  {
          e.printStackTrace();
       }
   }
}
```

17.2 Captura de audio

Otra aplicación frecuente es la captura de audio a través de este *API*. Para ello se debe implementar el siguiente código:

```
AudioFormat audioFormat = new AudioFormat(AudioFormat.LINEAR);
Vector dispositivos = CaptureDeviceManager.
getDeviceList(audioFormat);
CaptureDeviceInfo device = (CaptureDeviceInfo)dispositivos.
firstElement();
MediaLocator mediaLocator = device.getLocator();
Player player = Manager.createPlayer(mediaLocator);
player.realize();
player.start();
```

El código anterior permite crear un formato de audio lineal, con el cual se pueden capturar los dispositivos de audio del equipo. Con base en este conjunto de dispositivos, se selecciona el primero y con él, se crea un *MediaLocator* el cual permite la creación de un *Player*. El método *start* del *player* inicia la captura de audio que se reproducirá a través de los parlantes del equipo.

```
package Logica;

import java.util.Vector;
import javax.media.CaptureDeviceInfo;
import javax.media.CaptureDeviceManager;
import javax.media.MediaLocator;
import javax.media.Player;
import javax.media.format.AudioFormat;
import javax.media.Manager;

public class Audio {
   private Vector dispositivos;
   private CaptureDeviceInfo device = null;
   private AudioFormat audioFormat=null;
   private Player player = null;
   private MediaLocator mediaLocator = null;
```

```
    public Audio(){
        this.audioFormat = new AudioFormat(AudioFormat.LINEAR);
        this.dispositivos = CaptureDeviceManager.getDeviceList(this.audioFormat);
        if(this.dispositivos.size()>0){
            this.device=(CaptureDeviceInfo)this.dispositivos.firstElement();
            this.mediaLocator = this.device.getLocator();
            try{
                this.player = Manager.createPlayer(this.mediaLocator);
                this.player.realize();
                this.player.start();
            }
            catch ( Exception e ) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Audio audio = new Audio();
    }
}
```

CAPÍTULO 18

Carga dinámica de clases (*Reflection*)

Java permite en tiempo de ejecución cargar clases e invocar métodos de estas clases, que se encuentran en librerías. En programación este concepto se denomina *reflection*.

Por ejemplo, si se deseara cargar la clase *String* de Java que se encuentra en el paquete **java.lang**, se puede hacer mediante el método *forName* de la clase *Class*.

```
try {
   Class class1 = Class.forName("java.lang.String");
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

Es necesario capturar la excepción *ClassNotFoundException,* porque la clase que intenta cargarse puede no existir.

Es importante tener en cuenta que es suficiente utilizar el método *forName*, si y solo si se desea cargar una clase que hace parte de las librerías configuradas para el proyecto, como son las librerías del *JRE* y las librerías referenciadas.

Una vez cargada la clase es necesario crear una instancia. Si se desea usar el constructor por defecto, se puede crear la instancia mediante el método *newInstance*. La implementación es la siguiente:

```
package reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Reflection {

   public static void main(String[] args) {
       try {
          Class class1 = Class.forName("java.lang.String");
          Constructor []constructors = class1.getConstructors();
          Object obj = class1.newInstance();
       } catch (ClassNotFoundException e) {
          e.printStackTrace();
       } catch (IllegalArgumentException e) {
          e.printStackTrace();
       } catch (InstantiationException e) {
          e.printStackTrace();
       } catch (IllegalAccessException e) {
          e.printStackTrace();
       }
   }
}
```

Si la instancia se desea crear a través de la invocación de un constructor, es necesario capturar los constructores de la clase mediante el método *getConstructors*. El siguiente código captura e imprime los constructores de la clase *String*.

```
package reflection;

import java.lang.reflect.Constructor;

public class Reflection {

   public static void main(String[] args) {
       try {
          Class class1 = Class.forName("java.lang.String");
          Constructor []constructors = class1.getConstructors();
          for (int i=0; i<constructors.length; i++){
              System.out.println(constructors[i]);
          }
       } catch (ClassNotFoundException e) {
          e.printStackTrace();
       }

   }
}
```

Salida estándar

```
public java.lang.String()
public java.lang.String(java.lang.String)
public java.lang.String(char[])
public java.lang.String(char[],int,int)
public java.lang.String(int[],int,int)
public java.lang.String(byte[],int,int,int)
public java.lang.String(byte[],int)
public java.lang.String(java.lang.StringBuilder)
public java.lang.String(byte[],int,int,java.lang.String)
throws java.io.UnsupportedEncodingException
public java.lang.String(byte[],int,int,java.nio.charset.
Charset)
public java.lang.String(byte[],java.lang.String) throws java.
io.UnsupportedEncodingException
public java.lang.String(byte[],java.nio.charset.Charset)
public java.lang.String(byte[],int,int)
public java.lang.String(byte[])
public java.lang.String(java.lang.StringBuffer)
```

Una vez cargados los constructores se puede crear una instancia usando un constructor específico. Por ejemplo, si se deseara usar el tercer constructor que recibe por parámetro un arreglo tipo *char*, se invoca dicho método.

```
package reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Reflection {

   public static void main(String[] args) {
       try {
          Class class1 = Class.forName("java.lang.String");
          Constructor []constructors = class1.getConstructors();
          char []texto = {'H', 'o', 'l', 'a', ' ',
          'M', 'u', 'n', 'd', 'o'};
          Object obj  = constructors[2].newInstance(texto);
          System.out.println(obj);
       } catch (ClassNotFoundException e) {
          e.printStackTrace();
       } catch (IllegalArgumentException e) {
         e.printStackTrace();
       } catch (InstantiationException e) {
          e.printStackTrace();
       } catch (IllegalAccessException e) {
```

```
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Salida estándar
`Hola Mundo`

Una vez creada la instancia se puede hacer uso de los métodos de la clase. Para esto, es necesario crear un objeto que sea instancia de la clase *Method* a la que se le asigna el resultado del método *getMethod* de la clase *Class*. El método *getMethod* recibe por parámetro el nombre del método que se desea invocar y un arreglo con los tipos de datos de los parámetros de dicho método.

Con base en el ejemplo anterior, donde el objeto *obj* contiene un *string* con la información "*Hola Mundo*" y se desea invocar el método *split* de la clase *String*, con el fin de separar las palabras "*Hola*" y "*Mundo*", se debe crear el objeto *method* y luego usar el método *invoke*, el cual recibe por parámetro el objeto que contiene la información que en este caso es *obj* y un arreglo tipo *Object*, con los datos que se desean enviar por parámetro que, en este caso, solo es un espacio porque este es el carácter que separa las dos palabras. La implementación es la siguiente:

```
package reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Reflection {

   public static void main(String[] args) {
       try {
          Class class1 = Class.forName("java.lang.String");
          Constructor []constructors = class1.getConstructors();
          char []texto = {'H', 'o', 'l', 'a', ' ',
          'M', 'u', 'n', 'd', 'o'};
```

```
            Object obj  = constructors[2].newInstance(texto);
            System.out.println(obj);
            Method method = class1.getMethod("split", new Class[]{
            String.class});
            Object obj2 = method.invoke(obj, new Object[]{" "});
            String []palabras = (String [])obj2;
            System.out.println(palabras[0]);
            System.out.println(palabras[1]);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
}
```

Salida estándar

```
Hola Mundo
Hola
Mundo
```

El método *invoke* retorna un *object*, sin embargo, se sabe que el método *split* retorna un arreglo tipo *String*; por consiguiente, este *object* debe ser convertido a un arreglo tipo *String* mediante un *casting*. De esta manera, en el ejemplo el arreglo tipo *String* denominado "*palabras*" contiene el resultado del método *invoke*.

18.1 Carga dinámica mediante librerías

En caso que se desee cargar una clase que se encuentra en una librería externa, es necesario antes de cargar la clase cargar la librería, usando la clase *URLClassLoader*.

Considerando dos proyectos de ejemplo denominados MiLibreria1 y MiLibreria2, en donde el primer proyecto contiene un paquete denominado prueba1 con las clases *MiClase1* y *MiClase2*; y el segundo proyecto contiene un paquete denominado prueba2 con las clases *MiClase1* y *MiClase2*. La estructura de estos proyectos se presenta en la Figura 93.

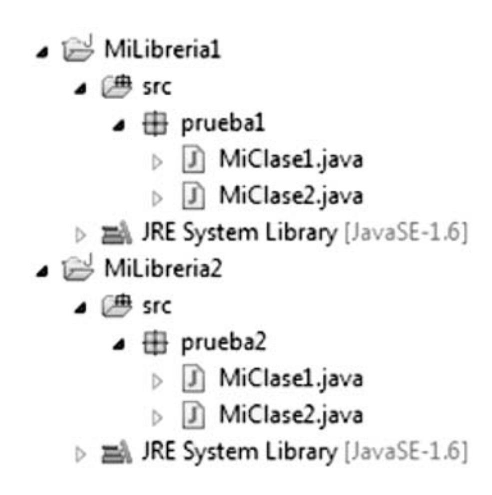


Figura 93. Proyectos de ejemplo para cargar con URLClassLoader

El contenido de las clases son:

Clase *prueba1.MiClase1*

```
package prueba1;

public class MiClase1 {

   public String metodo1(){
       return "Llamado a método 1 de la librería 1";
   }
}
```

Clase *prueba1.MiClase2*

```
package prueba1;

public class MiClase2 {

   public String metodo2(int a){
       return "Llamado a método 2 de la librería 1. " +
          "Parámetro entero a = " + a;
   }
}
```

Clase *prueba2.MiClase1*

```
package prueba2;

public class MiClase1 {

   public String metodo1(String a, int b){
       return "Llamado a método 1 de la librería 2. " +
          "Parámetros String a = " + a + "; " +
          "int b = " + b;
   }
}
```

Clase *prueba2.MiClase2*

```
package prueba2;

public class MiClase2 {

   public String metodo2(String a, double b, boolean c){
       return "Llamado a método 2 de la librería 2. " +
          "Parámetros String a = " + a + "; " +
          "double b = " + b + "; " +
          "boolean c = " + c;
   }
}
```

Los dos proyectos se exportan como archivo *jar*. Considerando un nuevo proyecto que requiere cargar las clases de las dos librerías denominadas *MiLibreria1.jar* y *MiLibreria2.jar*, se ubican estas librerías en una carpeta del proyecto (por ejemplo, *lib*).

Se debe crear una clase que herede de la clase *URLClassLoader*. En esta clase se cargan las librerías que se ubican en la carpeta *lib*, mediante el método *addURL*. Esta funcionalidad podría ubicarse en el constructor de la clase. En un método adicional se carga la clase que se requiere mediante el método *loadClass*. Posteriormente, se crea la instancia mediante el método *newInstance* de la clase *Class*. Luego se crea un objeto que sea instancia de la clase *Method* al que se le asigna el resultado del método *getMethod* de la clase *Class*. Finalmente se usa el método *invoke* de la clase *Method* para invocar el método requerido que está ubicado en una de las clases, de una de las librerías. Una posible implementación de esta clase es la siguiente:

Clase *MiClassLoader*

```
package reflection.classLoader;

import java.io.File;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class MiClassLoader extends URLClassLoader {

   public MiClassLoader(){
       super(new URL[]{});
       try {
          File folder = new File("lib/");
          for(String jarName : folder.list()){
              if(jarName.endsWith(".jar")){
                 this.addURL(new File("lib/"+jarName).toURL());
              }
          }
       } catch (MalformedURLException e) {
          e.printStackTrace();
       }
   }

   public Object ejecutarMetodo(String nombreClase,
   String nombreMetodo, Object []parametros, Class []
   tiposDeParametros){
       Class miClase;
```

```
        Object objResultado = null;
        try {
            miClase = this.loadClass(nombreClase);
            Object instance = miClase.newInstance();
            Method method = miClase.getMethod(
            nombreMetodo, tiposDeParametros);
            objResultado = method.invoke(instance, parametros);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return objResultado;
    }

    public static void main(String[] args) {
        MiClassLoader classLoader = new MiClassLoader();
        Object []parametros = {};
        Class []tiposDeParametros = {};
        Object objResultado;
        String resultado;
        //Llamado al metodo 1 de la libreria 1 ubicado en
        MiClase1
        objResultado = classLoader.ejecutarMetodo("prueba1.
        MiClase1", "metodo1", parametros, tiposDeParametros);
        resultado = (String)objResultado;
        System.out.println(resultado);

        //Llamado al metodo 2 de la libreria 1 ubicado en
        MiClase2
        parametros = new Object[1];
        parametros[0] = 5;
        tiposDeParametros = new Class[1];
        tiposDeParametros[0] = int.class;
        objResultado = classLoader.ejecutarMetodo("prueba1.
        MiClase2", "metodo2", parametros, tiposDeParametros);
        resultado = (String)objResultado;
        System.out.println(resultado);
```

```
        //Llamado al metodo 1 de la libreria 2 ubicado en
        MiClase1
        parametros = new Object[2];
        parametros[0] = "Diez";
        parametros[1] = 10;
        tiposDeParametros = new Class[2];
        tiposDeParametros[0] = String.class;
        tiposDeParametros[1] = int.class;
        objResultado = classLoader.ejecutarMetodo("prueba2.
        MiClase1", "metodo1", parametros, tiposDeParametros);
        resultado = (String)objResultado;
        System.out.println(resultado);

        //Llamado al metodo 2 de la libreria 2 ubicado en
        MiClase2
        parametros = new Object[3];
        parametros[0] = "Tres punto cinco";
        parametros[1] = 3.5;
        parametros[2] = true;
        tiposDeParametros = new Class[3];
        tiposDeParametros[0] = String.class;
        tiposDeParametros[1] = double.class;
        tiposDeParametros[2] = boolean.class;
        objResultado = classLoader.ejecutarMetodo("prueba2.
        MiClase2", "metodo2", parametros, tiposDeParametros);
        resultado = (String)objResultado;
        System.out.println(resultado);
    }
}
```

En el constructor de esta clase se cargan todos los archivos con extensión *jar*, los cuales corresponden a las librerías, mediante el siguiente fragmento de código:

```
File folder = new File("lib/");
for(String jarName : folder.list()){
   if(jarName.endsWith(".jar")){
       this.addURL(new File("lib/"+jarName).toURL());
   }
}
```

El método *ejecutarMetodo,* el cual recibe por parámetro el nombre de la clase que se desea cargar, el nombre del método que se desea invocar, los parámetros que se requieren enviar al método y los tipos de parámetros que tiene el método, tiene las siguientes

responsabilidades:

- Cargar la clase. Se usa el método *loadClass* y el nombre de la clase a cargar debe incluir el paquete.
- Crear la instancia. Se realiza mediante el método *newInstance.*
- Crear el método. Se usa el método *getMethod* de la clase *Class*, el cual recibe por parámetro en nombre del método y los tipos de parámetros que usa el método.
- Invocar el método. Se usa el método *invoke,* el cual recibe por parámetros la instancia creada y los parámetros requeridos por el método.

Para probar el concepto se han creado cuatro diferentes pruebas:

1. Invocar el método "*metodo1*" de la clase "*prueba1.MiClase1*". Este método no recibe parámetros y retorna un *String*. El código es el siguiente:

```
objResultado = classLoader.ejecutarMetodo("prueba1.MiClase1",
"metodo1", parametros, tiposDeParametros);
resultado = (String)objResultado;
System.out.println(resultado);
```

Salida estándar
`Llamado a método 1 desde la librería 1`

2. Invocar el método "*metodo2*" de la clase "*prueba1.MiClase2*". Este método recibe un parámetro tipo *int* y retorna un *String*. El código es el siguiente:

```
parametros = new Object[1];
parametros[0] = 5;
tiposDeParametros = new Class[1];
tiposDeParametros[0] = int.class;
objResultado = classLoader.ejecutarMetodo("prueba1.MiClase2",
"metodo2", parametros, tiposDeParametros);
```

```
resultado = (String)objResultado;
System.out.println(resultado);
```

Salida estándar
Llamado a método 2 desde la librería 1. Parámetro entero a = 5

3. Invocar el método "*metodo1*" de la clase "*prueba1.MiClase2*". Este método recibe un parámetro tipo *String*, un parámetro tipo *int* y retorna un *String*. El código es el siguiente:

```
parametros = new Object[2];
parametros[0] = "Diez";
parametros[1] = 10;
tiposDeParametros = new Class[2];
tiposDeParametros[0] = String.class;
tiposDeParametros[1] = int.class;
objResultado = classLoader.ejecutarMetodo("prueba2.MiClase1",
"metodo1", parametros, tiposDeParametros);
resultado = (String)objResultado;
System.out.println(resultado);
```

Salida estándar
Llamado a método 1 desde la librería 2. Parámetros String a = Diez; int b = 10

4. Invocar el método "*metodo2*" de la clase "*prueba1.MiClase2*". Este método recibe un parámetro tipo *String*, un parámetro tipo *double*, un parámetro tipo *boolean* y retorna un *String*. El código es el siguiente:

```
parametros = new Object[3];
parametros[0] = "Tres punto cinco";
parametros[1] = 3.5;
parametros[2] = true;
tiposDeParametros = new Class[3];
tiposDeParametros[0] = String.class;
tiposDeParametros[1] = double.class;
tiposDeParametros[2] = boolean.class;
objResultado = classLoader.ejecutarMetodo("prueba2.
MiClase2", "metodo2", parametros, tiposDeParametros);
```

```
resultado = (String)objResultado;
System.out.println(resultado);
```

Salida estándar
`Llamado a método 2 desde la librería 2. Parámetros String a = Tres punto cinco; double b = 3.5; boolean c = true`

18.2 Ejercicios propuestos

1. Cree una aplicación que permita cargar dinámicamente clases que contienen comportamiento equivalente a un robot. Esta aplicación debe emular el proyecto *Robocode*[1]. Esta aplicación debe tener las siguientes características:

    a. La aplicación debe manejar la graficación de los robots.

    b. Las clases que contienen comportamiento de robots puede avanzar el robot, rotar el robot, rotar el cañón y disparar.

    c. Debe permitirse la carga dinámica de mínimo dos robots y máximo ocho robots.

    d. La aplicación debe controlar el nivel de energía de cada robot y el número de balas. Si el nivel de energía de un robot es cero, el robot es destruido. Si solo queda un robot, este robot es el ganador.

    e. Debe haber una interfaz gráfica que permita la carga de los robots mediante el uso de cuadros de diálogo.

    f. Debe haber la opción de equipos de robots, en donde cada equipo debe tener dos robots. Cada robot es controlado por una clase diferente. Pueden ser cargados hasta cuatro equipos.

[1] http://robocode.sourceforge.net/

BIBLIOGRAFÍA

BISHOP, Judy. (1999). *Java Fundamentos de Programación.* Segunda Edición. Addison-Wesley.

CEBALLOS, Francisco. (2006). *Java 2.* Tercera Edición.

DEITEL & DEITEL. (2008). *Como programar en Java.* Séptima Edición.

ECKEL, Bruce. (1995) *Thinking in Java.*

GARCÍA, Javier; RODRÍGUEZ, José; MINGO, Iñigo; IMAZ, Aitor; BRAZÁLEZ, Alfonso; LARZABAL, Alberto; CALLEJA, Jesús; GARCÍA, Jon. (2000). *Aprenda Java como si estuviera en primero.* Universidad de Navarra.

GONZÁLEZ, Inmaculada; SANCHEZ, Antonio; HERNANDEZ, David. (2002). *Java Threads*. Departamento de Informática y Automática, Universidad de Salamanca.

INFOGRAFÍA

Eclipse Documentation. Tomado de *http://www.eclipse.org/documentation/*

JavaPlatform, Standard Edition 6. API Specification. Tomado de *http://docs.oracle.com/javase/6/docs/api/*

JFreeChart. Tomado de *http://www.jfree.org/*

Jigloo. Tomado de *http://www.cloudgarden.com/jigloo/*

MySql. Tomado de *http://dev.mysql.com/*

Oracle Technology Network. Java SE Technologies. Tomado de *http://www.oracle.com/technetwork/java/index.html*

Robocode. Tomado de *http://robocode.sourceforge.net/*

Xampp. Tomado de *http://www.apachefriends.org/es/xampp.html*

GLOSARIO

Abstracción. Capacidad que tiene un objeto para cumplir sus funciones independientemente del contexto en el que se utilice.

Algoritmo. Procedimiento o fórmula para resolver un problema.

AND. Operación lógica que coloca en la salida el valor verdadero o uno lógico si todas sus entradas tienen valor verdadero.

API. ***Application Programming Interface***. Interfaz de Programación de Aplicaciones usada por una aplicación para gestionar generalmente, servicios de bajo nivel, realizados por el sistema operativo del computador. Uno de los principales propósitos de una *API* consiste en proporcionar un conjunto de funciones de uso general adquiriendo beneficios en el proceso de desarrollo de *software.*

Aplicación. Cualquier programa de *software* que se ejecute en un sistema operativo y que haga una función específica para un usuario.

Applet. Pequeña aplicación escrita en Java, la cual se difunde a través de la red mediante la ejecución en el navegador *web* del cliente.

Archivo. Conjunto de datos que han sido codificados para ser manipulados por un computador.

Atributo. Elemento que hace referencia a una propiedad de una abstracción implementada en una clase.

Base de datos. Conjunto de datos que pertenecen al mismo contexto, almacenados sistemáticamente. En una base de datos, la información se organiza en campos y registros.

Bit. Símbolo binario.

Byte. Conjunto de 8 *bits*.

Chat. Sistema que permite conexiones en tiempo real y a menudo con carácter informal, entre dos o más personas, utilizando diversas aplicaciones o herramientas a través de una red.

Clase. Tipo Abstracto de dato que posee atributos y métodos.

Clic. Evento generado al oprimir alguno de los botones de un *mouse*. La palabra clic escrita se usa generalmente, para indicarle al usuario que oprima el botón del *mouse* sobre un área de la pantalla.

Cliente. Aplicación que permite a un usuario obtener un servicio de un servidor localizado en la red.

Código fuente. Conjunto de instrucciones que componen un programa, escrito en cualquier lenguaje.

Contraseña. Código utilizado para ingresar a un sistema restringido. Pueden contener caracteres alfanuméricos e incluso algunos otros símbolos. Una de las características importantes de la contraseña es que no es visible en la pantalla al momento de introducirla, con el propósito de que solo pueda ser conocida por el usuario.

Encapsulamiento. Principio del paradigma de orientación a objetos usado al desarrollar la estructura general de un programa mediante el cual cada componente posee visibilidad privada.

Extensión. Cadena de caracteres anexada al nombre de un archivo, antecedida por un punto y al final del nombre del archivo. Son usados para que el computador pueda reconocer fácilmente los archivos y usar los programas asociados a sus extensiones para abrirlos y manipularlos.

Herencia. Característica del paradigma de orientación a objetos en donde una clase puede acceder a los atributos y métodos de la clase padre o superclase.

HTML. *Hypertext Markup Language.* Es un lenguaje para crear documentos de hypertexto para uso mediante un explorador web.

HTTP. *Hypertext Transfer Protocol.* Protocolo de Transferencia de Hipertexto. *HTTP* es un protocolo para distribuir y manejar sistemas de información hipermedia.

Ícono. Símbolo gráfico que aparece en la pantalla con el fin de representar una determinada acción a realizar por el usuario.

Interfaz Gráfica de Usuario. Componente de una aplicación de *software* que el usuario visualiza y a través de la cual opera con ella. Está formada por ventanas, botones y menús, entre otros componentes

IP. *Internet Protocol.* Conjunto de reglas que regulan la transmisión de paquetes de datos a través de Internet. El *IP* es la dirección lógica de un computador en Internet de forma que cada dirección electrónica se asigna a un computador. La dirección *IP* está compuesta de cuatro octetos de *bits*.

J2ME. *Java 2 Micro Edition*. Versión Sun Microsystems de Java 2 destinada a dispositivos de recursos limitados como PDA, teléfonos móviles, sistemas electrónicos para vehículos, entre otros, requiriendo tan solo un mínimo de 128 Kb de RAM. Así, esta plataforma de Java está destinada a procesadores mucho menos potentes que los utilizados en los computadores.

Java. Lenguaje de programación orientada a objetos que permite ejecutar programas. Java fue originalmente desarrollado por Sun Microsystems y su principal objetivo fue crear un lenguaje que fuera capaz de ser ejecutado de una forma segura a través de Internet.

JavaScript. Lenguaje desarrollado por Sun Microsystems en conjunto con Netscape. El código Java script tiene como objetivo ser ejecutado en archivos *HTML.*

JDK. Java Development Kit. Componente básico para el desarrollo de *software* provisto por Sun Microsystems, que incluye las herramientas básicas necesarias para escribir, probar y depurar aplicaciones de Java.

JPEG, JPG. Formato de almacenamiento de imágenes que posee la gran ventaja de tener un formato comprimido que le permite ocupar poco espacio en la memoria

JRE. Java Runtime Environment. Componente requerido para que las aplicaciones desarrolladas en Java puedan ser ejecutadas en un computador

Kilobyte. Unidad de medida equivalente a 1024 *bytes*. Se usa frecuentemente para referirse a la capacidad de almacenamiento o tamaño de un archivo.

Megabyte. El *Megabyte* (*MB*) equivale a un millón de *bytes*, o mil *kilobytes* (exactamente 1,048,576 bytes). Hay 1024 Megabytes en un *Gigabyte*.

Método. Elemento que hace referencia a un servicio de una abstracción implementada en una clase.

Multimedia. Información digitalizada que combina texto, gráficos, imagen fija, imagen en movimiento y sonido.

MySQL. Sistema Gestor de Bases de Datos de *software* libre, considerado el más popular del mundo. Su ingeniosa arquitectura lo hace extremadamente rápido y fácil de personalizar.

Nibble. Conjunto de 4 *bits*.

NOT. Operación lógica que realiza la inversión de la entrada binaria.

Objeto. Referencia e instancia de una clase.

OR. Operación lógica que coloca en la salida el valor false o cero lógico si todas sus entradas tienen valor *false*.

OR Exclusiva XOR. Operación lógica presenta a la salida valor verdadero si el número de entradas verdaderas es impar.

Poilimorfismo. Característica del paradigma de orientación a objetos, basado del concepto de herencia, en donde permite que la referencia de una superclase tenga múltiples instancias. Esta situación permite que el objeto tenga múltiples comportamientos en tiempo de ejecución.

POO. **Programación Orientada a Objeto.** Concepción de la programación que organiza el desarrollo con base en la creación de objetos los cuales proveen propiedades y servicios del concepto abstraído.

Red. Sistema de comunicación de datos que conecta entre sí sistemas informáticos situados en lugares diferentes. Puede estar compuesta por diferentes combinaciones de diversos tipos de redes.

RGB. Modelo de color (*Red Green Blue*) utilizado para presentar color en los componentes gráficos de un sistema informático. Representa todos los colores como combinaciones de rojo, verde y azul.

Ruta absoluta. Ruta que parte del directorio raíz.

Ruta relativa. Ruta que parte del directorio donde se ejecuta la aplicación como origen. Esta ruta solo es relativa a un directorio.

Servidor. Computador que maneja peticiones de datos solicitados por otros computadores o dispositivos conectados en red. Un computador puede tener distintas aplicaciones de *software* de servidor, proporcionando muchos servicios a los clientes en la red.

Sistema Operativo. Programa especial el cual se carga en un computadoren el momento de encenderlo, y cuya función es gestionar los demás programas o aplicaciones que se ejecutarán en la máquina.

SQL. Structured Query Language. Es un lenguaje especializado de programación que permite realizar consultas a bases de datos. Los orígenes del *SQL* están ligados a los de las bases de datos relacionales.

Subclase. Clase que ha recibido características de otra clase mediante el uso del concepto de herencia.

Superclase. Clase que otorga características a otra clase mediante el uso del concepto de herencia.

Otros títulos de interés:

- **Lógica de programación**,
  Efraín M. Oviedo Regino

- **Redes locales (nivel básico)**,
  María Ángeles González Pérez

- **Seguridad informática**,
  Álvaro Gómez Vieites

- **Fundamentos de la prospectiva en sistemas de información,**
  Víctor Bañuls y José Salmerón

- **Openoffice.org2.x. Todo lo que necesita saber sobre el *software* libre,**
  Jorge E. Prieto H.

- **Hidráulica de ríos,**
  Tomás Ochoa

- **Diseño geométrico de carreteras,**
  James Cárdenas Grisales

- **Geometría descriptiva,**
  Germán Valencia García

Programación Orientada a Objetos usando Java



La siguiente obra presenta al lector, material concerniente al paradigma de Programación Orientada a Objetos, POO, mediante el lenguaje de programación Java. Así mismo, provee información acerca de los conceptos básicos de programación e historia del lenguaje Java.

En dieciocho capítulos describe la introducción al lenguaje de programación Java, conceptos fundamentales de programación, conceptos fundamentales de programación orientada a objetos, clases de utilidad del lenguaje Java, entrada y salida estándar, colecciones, manejo de archivos, concepto de herencia y polimorfismo, generación de documentación mediante *JavaDoc*, desarrollo de aplicaciones orientadas a arquitecturas, interfaces gráficas de usuario, conceptos fundamentales de computación gráfica, patrón modelo vista controlador, acceso a bases de datos, procesos multitarea, comunicaciones en red, manejo de multimedia y carga dinámica de clases.

Cada capítulo cuenta con un conjunto de problemas propuestos con el fin de que el lector tenga una herramienta adicional de análisis sobre los conceptos presentados.

Colección: Ingeniería y Arquitectura
Área: Informática



